LINUX

Průvodce jádrem operačního systému Linux

Linux je fenoménem Internetu. Vznikl jako zájmový studentský projekt a postupem času se stal nejoblíbenějším zdarma dostupným operačním systémem. Pro mnoho lidí to je záhada. Jak může zdarma dostupný produkt za něco stát? Jak může ve světě, v němž dominuje několik silných softwarových společností, obstát něco, co vytvořila parta „hackerů“? Jak může program, na jehož vývoji se podílí spousta lidí v řadě zemí, být stabilní a efektivní? Faktem ale je, že Linux obstál a opravdu je stabilní a efektivní. Ke své každodenní činnosti jej používá řada univerzit a výzkumných pracovišť. Řada lidí jej používá na svých domácích počítačích a vsadil bych se, že jej nějakým způsobem používá většina společností, i když si to ne vždy uvědomují. Linux se používá při procházení webem, jako hostitelský systém webovských sítí, k psaní článků a jako u všech počítačů a systémů také ke hraní her. Je třeba zdůraznit, že Linux není v žádném případě nějaká hračka - je to kvalitně navržený a profesionálně napsaný operační systém, používaný nadšenci na celém světě.

Kořeny Linuxu je možno vystopovat už od doby vzniku Unixu. V roce 1969 začal Ken Thompson, výzkumný pracovník Bell Laboratories, experimentovat s víceuživatelským a víceúlohovým operačním systémem na jinak nevyužitém počítači PDP-7. Záhy se k němu připojil Dennis Richie a spolu s dalšími pracovníky brzy vytvořili první verzi Unixu. Richie byl značně ovlivněn dřívějším projektem, systémem MULTICS, a samo jméno Unix je vlastně narážka na tento projekt. První verze byly napsány v assembleru, třetí verze systému však už byla přepsána do nového programovacího jazyka - jazyka C. Jazyk C Richie navrhl a napsal speciálně jako jazyk určený pro tvorbu operačních systémů. Tento přechod umožnil přenesení Unixu na podstatně výkonnější počítač PDP-11/45 a 11/70 společnosti Digital. Dál už to šlo rychle. Unix opustil prostředí laboratoří a výzkumných pracovišť a zanedlouho dodávala vlastní verze Unixu většina významných počítačových společností.

Linux vznikl jako z nouze ctnost. Linus Torvalds, autor Linuxu a tvůrce jeho návrhu, si mohl dovolit koupit pouze jediný program - Minix. Minix je velmi jednoduchý operační systém unixovského typu, často používaný jako výukový nástroj. Jeho schopnosti ovšem Linuse nijak nenadchly, takže se rozhodl pro vytvoření vlastního systému. Jako základní model si vybral Unix, s nímž byl jako student velmi dobře seznámen. Začal na počítači PC Intel 386 a pustil se do psaní. Šlo to velmi rychle a nadšený Linus nabídl výsledky své práce ostatním studentům pomocí celosvětové počítačové sítě, používané hlavně v akademickém prostředí. S programem se seznámili další lidé a začali k němu přispívat. Většina nově vzniklých kódů byla vlastně řešením nějakého problému, který měl jeho autor. Zanedlouho se Linux stal operačním systémem. Je třeba podotknout, že Linux neobsahuje žádný původní kód Unixu, představuje vlastní implementaci normy POSIX a používá velkou část kódu GNU (GNU není Unix) nadace Free Software Foundation, Cambridge, Massachusetts.

Řada lidí používá Linux jako jednoduchý nástroj; často jej pouze nainstalují z některého dobrého distribučního balíku CD-ROM. Většina uživatelů Linuxu jej používá k vytváření dalších aplikací nebo ke spouštění aplikací vytvořených někým jiným. Řada uživatelů Linuxu hltavě

čte dokumenty HOWTO,[1] pociťují jak nadšení, když se jim podaří nějakou část systému správně nakonfigurovat, tak i zklamání, když se jim to nepodaří. Pouze malá skupina uživatelů je schopna aktivně vytvářet ovladače zařízení a nabízet Linusi Torvaldsovi, který vytvořil a udržuje jádro Linuxu, návrhy na opravy jádra. Linus přijímá doplňky a úpravy jádra od kohokoliv. Může to znít jako spolehlivá cesta k anarchii, Linus však provádí velmi pečlivé kontroly a veškerý nový kód do jádra vkládá sám. Vždy existovala jen malá skupina lidí, kteří přispívali k jádru Linuxu.

Většina uživatelů Linuxu se nestará jak jejich systém funguje, ani jak drží pohromadě. Je to trochu škoda, protože studium Linuxu představuje velmi dobrý způsob, jak se naučit více o fungování operačních systémů. Nejen, že je Linux velmi dobře napsán, všechny jeho zdrojové kódy jsou navíc zdarma k dispozici. Je to dáno tím, že i když si autoři ponechávají autorská práva na své programy, svolili k jejich volné distribuci podle veřejné licence GNU nadace Free Software Foundation. Na první pohled mohou být zdrojové kódy matoucí - uvidíte adresáře jako `kernel`, `mm` a `net` - co ale obsahují a jak kód funguje? Je nutné hlubší pochopení celkové struktury a návrhu Linuxu. A to by mělo být cílem této knihy: jasně vysvětlit, jak operační systém Linux pracuje. Měli byste získat představu o tom, co se v systému děje, když například kopírujete soubor z jednoho místa na druhé, nebo když čtete elektronickou poštu. Dobře si vzpomínám na vzrušení, které jsem zažil, když jsem poprvé zjistil, jak operační systém skutečně pracuje. O toto vzrušení bych se chtěl podělit se čtenáři této knihy.

Můj zájem o Linux se datuje od roku 1994, kdy jsem navštívil Jima Paradise, který pracoval na přenosu Linuxu na platformu Alpha AXP. V té době jsem pracoval ve společnosti Digital Equipment Corporation. Od roku 1984 jsem působil převážně v oblasti sítí a komunikací, od roku 1992 jsem začal pracovat v nové divizi Digital Semiconductor. Cílem divize bylo uplatnit se na trhu procesorů a rozšířit procesory Alpha AXP a desky pro tyto procesory i mimo Digital. Když jsem o Linuxu poprvé slyšel, ihned jsem pochopil možnost, která se pro systémy Alpha otevírá. Jimovo nadšení bylo nakažlivé a začal jsem s ním na přenosu spolupracovat. Při této práci jsem musel stále více obdivovat nejen samotný operační systém, ale i jeho autory. Je to bezesporu pozoruhodná skupina lidí a to, že jsem se stal jejím členem, pro mne představovalo největší uspokojení za celou dobu, co pracuji na vývoji programů. Lidé se mne na Linux velmi často ptají jak v práci, tak doma a já velmi rád odpovídám. Čím více Linux používám jak v profesionálním, tak v osobním životě, tím více se stávám linuxovým nadšencem. Všimněte si, že jsem použil raději termín „nadšenec“ a ne „fanatik“; linuxového nadšence definuji jako člověka, který si je vědom skutečnosti, že existují i jiné operační systémy, raději je ale nepoužívá. Má manželka Gill, která používá Windows 95, svého času poznamenala: „Nikdy jsem si nemyslela, že budeme mít každý svůj operační systém.“ Mým potřebám inženýra Linux perfektně vyhovuje. Jedná se o vynikající, pružný a adaptabilní nástroj.

[1] Dokumenty HOWTO představují přesně to, co naznačuje jejich název - návod, jak něco udělat. Na téma Linuxu jich bylo napsáno velmi mnoho a všechny jsou užitečné.

Většina zdarma distribuovaných programů je k dispozici ve verzi pro Linux a často mi stačí pouze nahrát si předpřeložené spustitelné soubory nebo nainstalovat program z CD. Co jiného bych měl používat, abych se zadarmo naučil programovat v jazycích C++, Perl, nebo abych se dozvěděl něco o Javě?

Alpha AXP je pouze jednou z mnoha hardwarových platforem, na nichž Linux pracuje. Většina různých Linuxů pracuje na procesorech Intel, objevuje se však stále širší nabídka verzí pro jiné procesory. Sem patří Intel, MIPS, Sparc a PowerPC. Tuto knihu jsem mohl orientovat na kterýkoliv ze zmíněných procesorů, mé zkušenosti s Linuxem však vycházejí z verze pro procesor Alpha AXP, takže při ilustraci některých klíčových bodů se orientuji právě na tento procesor. Je však třeba říct, že 95 % zdrojového kódu jádra Linuxu je nezávislých na hardwarové platformě. Proto je i 95 % této knihy věnováno strojově nezávislým částem jádra Linuxu.

Komu je kniha určena

V této knize nedělám žádné předpoklady o znalostech a zkušenostech čtenářů. Věřím tomu, že zájem o problematiku je dostatečnou motivací k sebevzdělávání. Při čtení vám může pomoci jistá míra zkušenosti s počítači, nejlépe s počítači třídy PC, a také určitá znalost jazyka C.

Členění knihy

Tato kniha *není* psána jako dokumentace nitra Linuxu. Namísto toho se zabývá operačními systémy obecně a Linuxem zvláště. Jednotlivé kapitoly se řídí pravidlem „od obecného ke konkrétnímu“. Nejprve vždy obecně popisují činnost určitého subsystému jádra a poté se pouštějí do větších podrobností.

Záměrně jsem se rozhodl při popisu algoritmů jádra a metod, jak jádro řeší různé problémy, nepoužívat formulace jako `routine_X()` volá `routine_Y()`, která inkrementuje položku `foo` datové struktury `bar`. Takovéto věci zjistíte čtením kódu. Kdykoliv potřebuji vysvětlit část kódu nebo popsat něco podobného, začínám od čistého stolu popisem příslušných datových struktur. V knize je tedy poměrně podrobně popsána řada důležitých datových struktur jádra a jejich vzájemné vazby.

Jednotlivé kapitoly jsou poměrně nezávislé, stejně jako jednotlivé subsystémy jádra, které kniha popisuje. Někdy se však na sebe odkazují, protože například není možno popisovat procesy bez pochopení funkce virtuální paměti.

Kapitola *Základy hardwaru* představuje stručný popis moderního PC. Operační systém musí úzce spolupracovat s hardwarem, který představuje smysl jeho existence. Operační systém potřebuje některé služby, které mohou být zajištěny pouze hardwarově. Aby bylo možno operační systém Linux plně pochopit, je nutné základní porozumění hardwarové problematice.

Kapitola *Základy softwaru* představuje základní úvod do programování a zabývá se assemblerem a jazykem C. Popisuje nástroje používané při vytvoření operačního systému a obsahuje přehled funkcí a služeb operačního systému.

Kapitola *Správa paměti* popisuje způsob, jakým Linux obsluhuje fyzickou a virtuální paměť systému.

Kapitola *Procesy* popisuje co to proces je a jak jádro Linuxu vytváří, spravuje a ruší procesy v systému.

Při koordinaci své práce komunikují procesy navzájem mezi sebou a jádrem. Linux podporuje řadu mechanismů pro meziprocesovou komunikaci (*Inter-Process Communication Mechanism - IPC*). Dvěma z nich jsou signály a roury, Linux ale podporuje také mechanismy IPC Systemu V, pojmenované po verzi Unixu, v níž se poprvé objevily. Tyto mechanismy jsou popsány v kapitole *Meziprocesová komunikace*.

Standard PCI (*Peripheral Component Interconnect*) je dnes uznáván jako levný a výkonný standard pro datové sběrnice počítačů PC. Kapitola *PCI* popisuje jak jádro Linuxu inicializuje a používá PCI sběrnice a zařízení v systému.

Kapitola *Přerušení a jejich obsluha* se zabývá tím, jak jádro Linuxu obsluhuje přerušení. I když jádro obsahuje obecné mechanismy a rozhraní pro obsluhu přerušení, některé podrobnosti obsluhy závisejí na hardware a architektuře systému.

Jednou ze silných stránek Linuxu je podpora řady hardwarových zařízení, které jsou dnes k dispozici. Kapitola *Ovladače zařízení* popisuje, jak jádro Linuxu řídí fyzická zařízení připojená k systému.

Kapitola *Souborový systém* popisuje, jak jádro Linuxu spravuje soubory v souborových systémech, které podporuje. Popisuje virtuální souborový systém (VFS) a rovněž podporu reálných souborových systémů jádrem.

Sítě a Linux jsou dva pojmy s prakticky stejným významem. Linux je bez nadsázky produktem Internetu a WWW. Jeho autoři i uživatelé si pomocí webu vyměňují nápady a programy, a Linux sám bývá mnohde používán k zajištění síťových potřeb organizace. Kapitola *Sítě* popisuje, jak Linux podporuje síťové protokoly, označované společně termínem TCP/IP.

Kapitola *Mechanismy jádra* se zaměřuje na některé obecné úlohy a mechanismy, které musí jádro Linuxu zajišťovat, aby mohly jeho jednotlivé části účinně spolupracovat.

Kapitola *Moduly* popisuje, jak jádro dokáže dynamicky podle potřeby nahrávat různé funkce, například souborové systémy.

Kapitola *Zdrojový kód* vysvětluje, kde máte ve zdrojovém kódu Linuxu hledat konkrétní funkce.

Příloha *Datové struktury* obsahuje okomentovaný definiční kód datových struktur, které byly v předchozích kapitolách popsány.

V příloze *Procesor Alpha* jsou uvedeny hlavní charakteristiky architektury Alpha AXP.

Příloha *Užitečné adresy WWW a FTP* uvádí seznam adres, na nichž můžete prostřednictvím Internetu získat další informace o problematice Linuxu.

Konečně *Slovníček* představuje stručné vysvětlení hlavních pojmů, s nimiž se v této knize setkáte.

V textu knihy naleznete odkazy na hierarchii zdrojového kódu jádra Linuxu. Uvádím je pro případ, že byste se chtěli podívat přímo na zdrojový kód. Všechny odkazy jsou relativní vzhledem k cestě `/usr/src/linux`. Pokud vezmeme jako příklad soubor `foo/bar.c`, naleznete jej jako `/usr/src/linux/foo/bar.c`. Pokud používáte Linux (což byste měli), představuje studium zdrojového kódu cennou zkušenost a tuto knihu můžete použít jednak jako příručku pro pochopení kódu a jednak jako průvodce různými datovými strukturami.

Ochranné známky

Caldera, OpenLinux a logo „C“ jsou ochranné známky společnosti Caldera, Inc.

Caldera OpenDOS 1997 je ochranná známka společnosti Caldera, Inc.

DEC je ochranná známka společnosti Digital Equipment Corporation.

DIGITAL je ochranná známka společnosti Digital Equipment Corporation.

Linux je ochranná známka Linuse Torvaldse.

Motif je ochranná známka společnosti Open System Foundation, Inc.

MS-DOS je ochranná známka společnosti Microsoft Corporation.

Red Hat a logo Red Hat jsou ochranné známky společnosti Red Hat Software, Inc.

Unix je registrovaná ochranná známka společnosti X/Open.

XFree86 je ochranná známka společnosti XFree86 Project, Inc.

X Window System je ochranná známka společnosti X Consortium a Massachusetts Institute of Technology.

Poděkování

Musím poděkovat řadě lidí, kteří si udělali čas a poslali mi připomínky k této knize. Snažím se všechny připomínky zahrnovat do nových verzí. Zvláštní poděkování si zaslouží John Rigby a Michael Bauer, kteří důkladně přečetli a opravili celou knihu. Nebylo to nic jednoduchého.

Odkazy na zdrojové texty jádra

1. Viz `foo` v `foo/bar.c`

Kapitola

1

Základy hardwaru

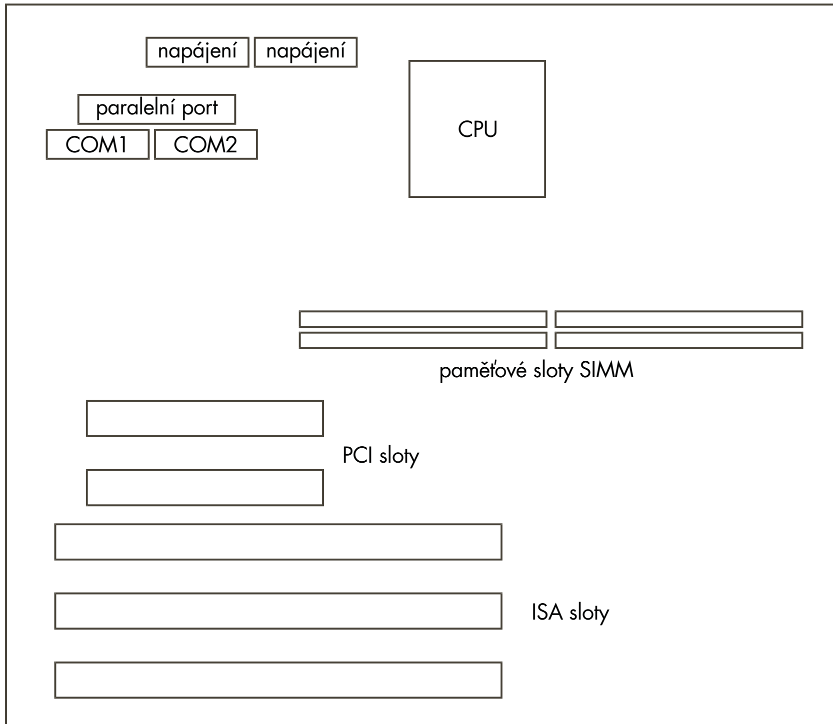
Operační systém musí úzce spolupracovat s hardwarem, který jej hostí. Operační systém potřebuje některé služby, které je možno zajistit pouze hardwarově. K úplnému pochopení činnosti operačního systému Linux je nutné porozumět základům hardwaru. Tato kapitola představuje stručný úvod do hardwaru moderního PC.

Revoluce začala v lednu roku 1975, kdy se na obálce časopisu „Popular Electronics" objevil obrázek počítače Altair 8800.

Počítač Altair 8800, pojmenovaný podle jednoho z prvních dílů seriálu Star Trek, byl počítač, který si mohl zájemce o elektroniku postavit za pouhých 397 dolarů. S procesorem Intel 8080 a 256 bajty paměti bez obrazovky či klávesnice to bylo podle dnešních standardů úplné nic. Jeho autor, Ed Roberts, pojmenoval svůj vynález „personal computer". V dnešní době se termínem PC označuje prakticky každý počítač, který můžete odnést v rukou. Podle této definice jsou počítači PC i některé velmi výkonné systémy Alpha AXP.

Různí nadšenci pochopili sílu Altairu a začali pro něj vytvářet programy a hardware. Pro tyto průkopníky představoval Altair svobodu - svobodu od dávkových úloh pro mainframové systémy, spravované elitou kněžstva. Na fenoménu počítače, který můžete mít doma na kuchyňském stole, vydělali závratné bohatství dva zběhlí vysokoškolští studenti. Objevila se řada různého hardwaru s různými rozdíly a softwaroví nadšenci radostně vytvářeli nové programy pro nové systémy. Paradoxně to byla společnost IBM, která ohlášením počítače IBM PC v roce 1981 (k zákazníkům se dostal v roce 1982) určila formu moderních osobních počítačů. S procesorem Intel 8088, 64 KB paměti (rozšiřitelnými na 256 KB), dvěma disketovými mechanikami a displejem Colour Graphics Adapter (CGA) s rozlišením 80 znaků na 25 řádků to podle dnešních měřítek nebylo nic moc, počítač se ale prodával velmi dobře. V roce 1983 následoval model IBM PC/XT, doplněný o neuvěřitelných 10 MB pevného disku. Krátce předtím vytvořily své klony architektury IBM PC i jiné společnosti, jako například Com-

paq, a architektura PC se tak stala de facto standardem. Tento standard umožnil, aby řada hardwarových společností začala soutěžit na stále se rozrůstajícím trhu, což vedlo ke snížení cen. Řada rysů architektury těchto prvních PC byla přenesena i do moderních PC. Například i ten nejvýkonnější systém s procesorem Intel Pentium Pro se spouští v adresačním režimu Intel 8086. Když Linus Torvalds začínal psát to, z čeho později vzniknul Linux, vybral si nejrozšířenější a cenově dostupnou platformu PC Intel 80386.



Obrázek 1.1
Základní deska typického PC

Když se na PC podíváme zvenčí, nejnápadnějšími komponentami je skříň počítače, klávesnice, myš a monitor. Na čelní stěně skříně jsou nějaká tlačítka, malý displej s nějakými čísly a disketová mechanika. Většina dnešních systémů má také mechaniku CD-ROM a pokud vám zaleží na vašich datech, můžete mít také páskovou jednotku pro zálohování. Všechna tato zařízení se společně označují termínem *periferní zařízení*.

Přestože celkové řízení systému má na starosti procesor (CPU), není to jediné inteligentní zařízení. Jistou úroveň inteligence mají i řadiče jednotlivých periferií, například řadič IDE. Uvnitř PC (viz obrázek 1.1) najdeme základní desku, která obsahuje procesor či CPU, paměť a několik slotů pro připojení periferií ISA nebo PCI. Některé řadiče, například řadič IDE disku, mohou být vestavěny přímo na základní desku.

1.1 Procesor

Procesor, CPU nebo mikroprocesor je srdcem každého počítačového systému. Procesor načítá a vykonává instrukce uložené v paměti a provádí tak operace matematické, logické a řídí toky dat. V začátcích počítačové éry byly jednotlivé části procesoru samostatné (a fyzicky velké) moduly. V té době vznikl termín *Central Processing Unit* (CPU). Moderní mikroprocesor kombinuje všechny tyto moduly v jednom integrovaném obvodu, vytvořeném na malinkém kousku křemíku. V této knize budeme chápat termíny *CPU*, *procesor* a *mikroprocesor* jako rovnocenné.

Mikroprocesor pracuje s binárními daty, tedy daty, složenými z nul a jedniček.

Jedničky a nuly jsou reprezentovány zapnutím nebo vypnutím elektrického napětí. Stejně jako desítkové číslo 42 znamená vlastně „4 desítky a 2 jedničky“, je i binární číslo posloupnost binárních číslic, z nichž každá reprezentuje určitou mocninu dvou. 10 na první (10^1) je 10, 10 na druhou (10^2) je 10×10, 10^3 je 10×10×10 a tak dále. Binárně 0001 je desítkově 1, binárně 0010 je desítkově 2, binárně 0011 je 3, 0100 je 4 a tak dále. Takže desítkových 42 je binárně 101010 nebo také 2 + 8 + 32 nebo $2^1 + 2^3 + 2^5$. V počítačových programech se ale čísla neuvádějí ve dvojkové soustavě, namísto toho se obvykle používá soustava šestnáctková.

Při základu šestnáct reprezentuje každý řád jednu mocninu šestnácti. Desítkové číslice jsou pouze 0 až 9, číslice 10 až 15 se jedním znakem zapisují jako písmena A, B, C, D, E a F. Například šestnáctkově E je desítkově 14, šestnáctkově 2A je desítkově 42 (dvě šestnáctky plus deset). Podle konvencí programovacího jazyka C (kterou v knize používám) se šestnáctková čísla uvozují znaky „*0x*“; šestnáctkových 2A se tedy zapisuje jako *0x2A*.

Mikroprocesor může provádět matematické operace jako sčítání, násobení a dělení a logické operace typu „je X větší než Y?“.

Činnost procesoru je řízena vnějšími hodinami. Tyto hodiny, takzvané systémové hodiny, generují pravidelné hodinové impulsy a s každým impulsem provede procesor nějakou operaci. S každým tikem hodin může procesor například vykonat jednu instrukci. Rychlost procesoru se udává rychlostí „tikání“ systémových hodin. Procesor s rychlostí 100 MHz dostává každou sekundu 100,000,000 hodinových impulsů. Výkon procesoru se ale nedá pomocí hodinového kmitočtu posuzovat, protože různé procesory stihnou za jeden hodinový impuls různé množ-

ství práce. Obecně ale platí, že čím rychlejší hodiny, tím výkonnější procesor. Instrukce vykonávané procesorem jsou velmi jednoduché, například „přečti obsah paměti na adrese X do registru Y“. Registry jsou interní pamětí mikroprocesoru, slouží k ukládání dat a k manipulaci s nimi. Prováděná operace může způsobit, že procesor přeruší to, co momentálně dělá, a odskočí na úplně jinou instrukci někde jinde v paměti. Jednotlivé jednoduché instrukce dávají moderním procesorům prakticky neomezené možnosti, protože mohou zpracovat milióny nebo dokonce miliardy takovýchto instrukcí za sekundu.

Před vykonáním instrukce je nutné ji načíst z paměti. Sama instrukce se může odkazovat na data uvnitř paměti, která se pak podle potřeby načítají nebo ukládají.

Velikost, počet a typ registrů závisí výhradně na typu procesoru. Procesor Intel 80486 má jinou sadu registrů než procesor Alpha AXP, první rozdíl může být ten, že registry procesoru Intel jsou 32bitové, registry procesoru Alpha jsou 64bitové. Obecně ale platí, že procesory mají větší počet univerzálních registrů a menší počet speciálním registrů. Většina procesorů má následující speciální, vyhrazené registry:

Ukazatel instrukcí (*Program Counter, PC*)

Tento registr obsahuje adresu instrukce, která se bude provádět v dalším kroku. Obsah registru PC se automaticky inkrementuje po každém načtení instrukce.

Ukazatel zásobníku (*Stack Pointer, SP*)

Procesory mají přístup k většímu objemu externí paměti s náhodným přístupem (RAM), která slouží k dočasnému ukládání dat. Zásobník představuje jednoduchou metodu jak do externí paměti ukládat a načítat dočasné hodnoty. Procesory jsou obvykle vybaveny instrukcemi, které umožňují uložit jednu hodnotu na zásobník a přečíst ji později zpět. Zásobník pracuje na principu „poslední dovnitř, první ven“ (*Last In First Out, LIFO*). Jinak řečeno, pokud uložíte na zásobník dvě hodnoty, X a Y, a poté vyzvednete ze zásobníku jednu hodnotu, dostanete hodnotu Y.

U některých procesorů se zásobník rozrůstá nahoru směrem k vrcholu paměti, u jiných postupuje shora dolů ke dnu paměti. Některé procesory, například ARM, podporují oba typy zásobníků.

Stavový registr (*Processor Status, PS*)

Instrukce mohou dávat nějaký výsledek, například instrukce „je obsah registru X větší než obsah registru Y“ vrací hodnotu „ano“ nebo „ne“. Tyto a další informace o momentálním stavu procesoru jsou uloženy v registru PS. Například většina procesorů má minimálně dva pracovní režimy, režim jádra a uživatelský režim. Registr PS obsahuje i informaci o momentálním režimu činnosti.

1.2 Paměť

Všechny systémy mají nějakou hierarchii paměti, na různých úrovních této hierarchie se nacházejí paměti s různou rychlostí a velikostí. Nejrychlejší paměť se označuje jako *cache* (skladiště, zásobárna) a její funkce přesně odpovídá jejímu jménu – slouží k dočasnému uložení obsahu hlavní paměti. Tato paměť je sice velmi rychlá, ale také velmi drahá, takže většina procesorů obsahuje malou interní cache a na základní desce je pak větší externí cache. Některé procesory používají stejnou paměť cache pro uložení instrukcí i dat, jiné mají dvě paměti - jednu pro instrukce, druhou pro data. Procesory Alpha AXP obsahují dvě interní paměti cache, jednu pro data (D-Cache) a druhou pro instrukce (I-Cache). Externí cache (B-Cache) je pro obě společná. Kromě toho systém ještě obsahuje hlavní paměť, která je v porovnání s externí pamětí cache velmi pomalá. V porovnání s interní cache procesoru se hlavní paměť přímo ploužÍ.

Obsah pamětí cache a hlavní paměti se musí shodovat, musí být *koherentní*. Jinak řečeno, pokud se nějaké slovo z hlavní paměti nachází v některé paměti cache, musí systém zajistit, aby obsah cache a hlavní paměti byl stejný. Zajištění koherence pamětí cache je částečně úkolem hardwaru, částečně úkolem operačního systému. To je ovšem společné i pro řadu jiných systémových úloh, kdy musí hardware a software při zajišťování nějaké činnosti úzce spolupracovat.

1.3 Sběrnice

Jednotlivé obvody na systémové desce jsou propojeny systémem, nazývaným sběrnice. Systémová sběrnice se dělí na tři logické celky, na adresovou sběrnici, datovou sběrnici a řídící sběrnici. Datová sběrnice udává místo v paměti (adresu) pro datový přenos. Datová sběrnice slouží k přenosu dat, je obousměrná a umožňuje načíst data do procesoru nebo je z procesoru zapsat. Řídící sběrnice obsahuje různé linky, které slouží k přenosu časovacích a řídících signálů pro celý systém. Existuje řada různých sběrnic, oblíbené sběrnice pro připojení periferií k systému jsou například sběrnice ISA a PCI.

1.4 Řadiče a periferie

Periferie jsou reálná zařízení, jako například grafické karty nebo disky, ovládané čipem řadiče, který může být na základní desce nebo na kartě, zapojené do základní desky. IDE disky jsou řízeny řadičem IDE, SCSI disky řídí řadič SCSI a podobně. Řadiče jsou připojeny k procesoru a k sobě navzájem různými sběrnicemi. Většina dnešních systémů používá k propojení hlavních komponent systému sběrnice ISA a PCI. Řadiče jsou rovněž procesory, podobně jako CPU, a je možno je chápat jako inteligentní pomocníky procesoru, který zajišťuje celkové řízení systému.

Každý řadič je jiný, obvykle však mají registry, jejichž prostřednictvím se ovládají. Program spuštěný na CPU musí být schopen do těchto registrů zapisovat a číst z nich. Jeden registr může například obsahovat příznak chyby. Další může být použit k různým řídicím účelům podle režimu řadiče. CPU může individuálně adresovat každý řadič v systému, což znamená, že musí existovat softwarový ovladač zařízení, který manipuluje s registry řadiče. Dobrým příkladem je sběrnice IDE, která umožňuje individuálně ovládat každý k ní připojený disk. Dalším příkladem může být sběrnice PCI, která umožňuje zvlášť ovládat každé k ní připojené zařízení (například grafickou kartu).

1.5 Adresový prostor

Systémová sběrnice spojuje CPU s hlavní pamětí, je to jiná sběrnice než ta, kterou jsou k CPU připojeny hardwarové periferie. Paměťový prostor, na němž jsou souhrnně připojeny periferie, se označuje jako V/V prostor. V/V prostor může být dále dělen, to nás ale v této chvíli nemusí zajímat. CPU může přistupovat jak k adresovému prostoru systémové paměti, tak k adresovému prostoru V/V zařízení, zatímco řadiče mohou k systémové paměti přistupovat pouze nepřímo, a to pouze s podporou procesoru. Z pohledu zařízení, řekněme ovladače disketových mechanik, je vidět pouze ta část adresového prostoru, v němž se zobrazují registry tohoto zařízení (ISA), není vidět systémová paměť. CPU má typicky odlišné instrukce pro přístup k paměti a k V/V prostoru. Může například existovat instrukce s významem „přečti bajt z V/V adresy *0x3f0* do registru X“. To je přesně způsob, jakým CPU řídí hardwarové periferie, čtením a zápisem registrů v V/V prostoru. Která část V/V prostoru je přidělena kterým periferiím (řadiči IDE, sériovým portům, řadiči disket a podobně) je dáno konvencí už od doby vzniku architektury PC. Adresa *0x3f0* z V/V prostoru je shodou okolností adresa řídícího registru jednoho ze sériových portů (COM1).

Jsou chvíle, kdy řadič potřebuje přečíst nebo zapsat větší objem dat přímo z nebo do systémové paměti. Příkladem může být zápis dat na pevný disk. V takovém případě se používá řadič přímého přístupu do paměti (Direct Memory Access, DMA), který umožňuje hardwarovým zařízením přímý přístup k systémové paměti, ovšem pouze pod přísným řízením a dohledem CPU.

1.6 Hodiny

Všechny operační systémy potřebují znát čas, takže moderní PC jsou vybaveny zvláštní periferií, zvanou hodiny reálného času (Real Time Clock, RTC). Tyto hodiny zajišťují dvě věci: informaci o přesném čase a generování pravidelných časových intervalů. RTC mají vlastní baterii, takže pracují, i když je počítač vypnut, proto tedy PC vždy „zná“ správné datum a čas. Intervalový časovač umožňuje operačnímu systému přesně plánovat základní úkoly.

Kapitola

2

Základy softwaru

Program je posloupnost počítačových instrukcí, které vykonávají nějakou úlohu. Program může být napsán v assembleru, počítačovém jazyce nízké úrovně, nebo v nějakém vyšším, na platformě nezávislém jazyce, jako je například C. Operační systém je speciální program, který uživatelům umožňuje spouštět aplikace jako jsou tabulkové procesory a textové editory. Tato kapitola představuje úvod do základních programovacích principů a vysvětluje úkol a funkci operačního systému.

2.1 Počítačové jazyky

2.1.1 Assemblery

Instrukce, které CPU načítá z paměti a provádí, nejsou pro člověka vůbec srozumitelné. Jedná se o strojový kód, který počítači přesně říká, co má dělat. Šestnáctkové číslo *0x89e5* je pro procesor Intel 80486 instrukce, která zkopíruje obsah registru ESP do registru EBP. Jedním z prvních programátorských nástrojů vyvinutých už pro první počítače byl assembler, program, který bere pro člověka srozumitelný zdrojový kód a přeloží jej do strojového kódu. Jazyk assembler provádí manipulace s registry a daty explicitně, je jiný pro každý mikroprocesor. Assembler pro procesory Intel *x*86 se výrazně liší od assembleru pro mikroprocesor Alpha AXP. Následující ukázka assembleru procesoru Alpha AXP ukazuje, co může takový program dělat:

```
    ldr r16, (r15)        ; řádek 1
    ldr r17, 4(r15)       ; řádek 2
    beq r16,r17,100       ; řádek 3
    str r17, (r15)        ; řádek 4
100:                      ; řádek 5
```

První příkaz (na řádku 1) naplní registr 16 hodnotou na adrese, uložené v registru 15. Druhý příkaz naplní registr 17 hodnotou z následující adresy. Třetí příkaz porovná hodnotu registrů 16 a 17 a pokud se shodují, skočí na návěští *100*. Pokud registry neobsahují stejnou hodnotu, pokračuje se řádkem 4, na němž se obsah registru 17 uloží do paměti. Pokud registry obsahují stejnou hodnotu, není nutné ji ukládat. Programy v assembleru se píší obtížně a zdlouhavě, navíc se v nich snadno dělají chyby. Pouze velmi malá část jádra Linuxu je napsána přímo v assembleru. Tyto části jsou závislé na konkrétním procesoru a důvodem použití assembleru je dosažení maximálního výkonu.

2.1.2 Programovací jazyk C a kompilátor

Psát v assembleru větší programy je obtížné a zdlouhavé. Často vznikají chyby a výsledné programy nejsou přenositelné, protože jsou vázány na určitou rodinu procesorů. Daleko lepší je použít strojově nezávislý jazyk, jako je například C. Jazyk C umožňuje popisovat program pomocí jeho logického algoritmu a pomocí dat, nad nimiž operuje. Speciální programy zvané kompilátory čtou program v jazyce C a překládají jej do assembleru, generují z něj konkrétní strojový kód. Dobrý kompilátor dokáže překládat tak, že výsledný kód je téměř stejně efektivní jako kdyby byl napsán zkušeným programátorem přímo v assembleru. Většina jádra Linuxu je napsána v jazyce C. Vezměme si následující příkaz jazyka C:

```
if (x != y)
        x = y ;
```

Tento příkaz provádí přesně stejnou operaci jako výše uvedený příklad v assembleru. Pokud se obsah proměnné `x` neshoduje s obsahem proměnné `y`, překopíruje se obsah proměnné `y` do `x`. Program v jazyce C je organizován do rutin, které provádějí jednotlivé funkce. Každá rutina může vracet jakoukoliv hodnotu nebo datový typ podporovaný jazykem C. Velké programy jako například jádro Linuxu se mohou skládat z mnoha samostatných C-modulů, každý z nich má své vlastní rutiny a datové struktury. Jednotlivé moduly zdrojového kódu plní příbuzné logické funkce, jako například obsluhu souborového systému.

Jazyk C podporuje mnoho typů proměnných; proměnná je místo v paměti, na něž se dá odkazovat symbolickým jménem. Ve výše uvedeném fragmentu programu označují `x` a `y` místa v paměti. Programátor se nestará o to, kam v paměti budou data uložena, to je záležitostí linkeru (viz dále). Některé proměnné obsahují různé typy dat, jako například celá nebo reálná čísla, jiné představují ukazatele.

Ukazatel je proměnná, která obsahuje adresu - místo v paměti, kde jsou uložena jiná data. Představme si proměnnou `x`. Ta může být v paměti uložena na adrese *0x80010000*. Můžete mít ukazatel, řekněme `px`, který bude ukazovat na `x`. Ukazatel `px` může „bydlet" na adrese *0x80010030*. Hodnota ukazatele `px` bude *0x8001000*, tedy adresa proměnné `x`.

Jazyk C umožňuje shromažďovat do jedné datové struktury spolu související proměnné. Například takto:

```
struct {
        int i ;
        char b ;
} my_struct ;
```

Toto je datová struktura nazvaná `my_struct`, která obsahuje dva prvky, celé číslo (32 bitů) zvané `i` a znak (8 bitů) zvaný `b`.

2.1.3 Linker

Linker je program, který „slinkuje" dohromady několik modulů a knihoven a výsledkem je jediný program. Moduly obsahují strojový kód vygenerovaný assemblerem nebo překladačem a najdeme v nich jednak spustitelný kód a data a dále informace, které říkají linkeru, jak jednotlivé moduly zkombinovat do výsledného programu. Jeden modul může například obsahovat všechny databázové funkce programu, jiný modul může obsahovat funkce pro obsluhu řádkových příkazů. Linker propojí odkazy mezi těmito moduly v místech, kde se rutina nebo data, na něž se jeden modul odkazuje, nacházejí v modulu jiném. Jádro Linuxu je jediný velký program, vzniklý slinkováním řady modulů.

2.2 Co je to operační systém

Bez příslušného softwaru je počítač jenom hromada elektroniky, která vydává teplo. Pokud je hardware srdcem počítače, je software jeho duší. Operační systém je soubor systémových programů, které uživateli umožňují spouštět aplikační programy. Operační systém abstrahuje od skutečného hardwaru systému a přináší uživatelům a aplikacím virtuální stroj. V zásadě platí, že teprve operační systém představuje charakteristiku počítače. Většina uživatelů PC používá jeden nebo více operačních systémů, z nichž každý může mít úplně jiný vzhled a chování. Linux je tvořen řadou samostatných dílů, které dohromady tvoří operační systém. Nutným dílem Linuxu je jeho jádro, ani to by však nebylo k ničemu bez knihoven nebo příkazových interpretů.

Abychom si lépe ukázali, co to operační systém vlastně je, zkusme si představit co se stane, když zadáte jednoduchý příkaz:

```
$ ls
Mail            c               images          perl
docs            tcl
$
```

Znak $ je prompt přihlašovacího příkazového interpretu (v tomto případě bash). Znamená to, že se čeká na vás, na uživatele, až zadáte nějaký příkaz. Když zadáváte příkaz ls, ovladač klávesnice rozpozná, že jsou zadávány nějaké znaky. Ovladač klávesnice je předá příkazovému interpretu, který příkaz zpracuje tak, že se pokouší najít spustitelný soubor stejného jména. Najde jej jako soubor /bin/ls. Zavolají se služby jádra, které přemístí kód obsažený v souboru do virtuální paměti a zahájí jeho provádění. Příkaz ls volá souborové služby jádra a zjišťuje, jaké soubory existují. Souborový systém může využít informací uložených ve vyrovnávací paměti souborového systému, nebo použije ovladač diskového zařízení a načte údaje z disku. Může dojít i k tomu, že síťový ovladač začne komunikovat se vzdáleným počítačem aby zjistil podrobnosti o vzdálených souborech, k nimž má váš systém přístup (souborové systémy je možné vzdáleně připojit pomocí služby Network File System, NFS). Ať už se informace zjistí jakýmkoliv způsobem, ls je vypíše a ovladač displeje je zobrazí na obrazovce.

Celé to vypadá dost komplikovaně, nicméně takovýto jednoduchý příklad nám ukazuje, že operační systém je ve skutečnosti řada spolupracujících funkcí, které dohromady dávají uživateli souvislý pohled na systém.

2.2.1 Správa paměti

Pokud bychom měli neomezené prostředky, řekněme paměť, řada funkcí, které operační systém provádí, by byla zbytečných. Jedním ze základních triků každého operačního systému je zajistit, aby se malý objem fyzické paměti choval jako paměť daleko větší. Této obrovské paměti říkáme virtuální paměť. Vtip je v tom, že programy, které v systému běží, se domnívají, že mají k dispozici celou tuto velikou paměť. Systém v době běhu rozděluje paměť na snadno manipulovatelné stránky a tyto stránky podle potřeby odkládá na disk. Programy si ničeho nevšimnou díky dalšímu triku, multiprocessingu.

2.2.2 Procesy

Proces můžeme chápat jako spuštěný program. Každý proces je samostatná entita, která vykonává nějaký program. Pokud se podíváte na procesy v Linuxu, uvidíte jich celou řadu. Například zadáním příkazu ps uvidíte následující seznam procesů:

```
$ ps
  PID TTY STAT  TIME COMMAND
  158 pRe 1     0:00 -bash
  174 pRe 1     0:00 sh /usr/X11R6/bin/startx
  175 pRe 1     0:00 xinit /usr/X11R6/lib/X11/xinit/xinitrc --
  178 pRe 1 N   0:00 bowman
  182 pRe 1 N   0:01 rxvt -geometry 120x35 -fg white -bg black
```

```
  184 pRe 1 <  0:00 xclock -bg grey -geometry -1500-1500 -padding 0
  185 pRe 1 <  0:00 xload -bg grey -geometry -0-0 -label xload
  187 pp6 1    9:26 /bin/bash
  202 pRe 1 N  0:00 rxvt -geometry 120x35 -fg white -bg black
  203 ppc 2    0:00 /bin/bash
 1796 pRe 1 N  0:00 rxvt -geometry 120x35 -fg white -bg black
 1797 v06 1    0:00 /bin/bash
 3056 pp6 3 <  0:02 emacs intro/introduction.tex
 3270 pp6 3    0:00 ps
$
```

Pokud by můj systém měl mnoho procesorů, mohl by každý proces (přinejmenším teoreticky) běžet na jednom procesoru. Bohužel ve skutečnosti mám pouze jediný procesor, takže operační systém se uchyluje k dalším trikům a nechává každý proces běžet pouze krátkou dobu. Tento čas se označuje jako časové kvantum. Celému triku se říká multiprocessing nebo také plánování a způsobuje, že každý proces si myslí, že on je jediný proces v systému. Procesy jsou jeden před druhým chráněny, takže když jeden proces zhavaruje nebo začne pracovat chybně, neovlivní to nijak ostatní procesy. Operační systém toho dosahuje tím, že každému procesu přidělí samostatný adresový prostor a proces může manipulovat pouze s ním.

2.2.3 Ovladače zařízení

Ovladače zařízení představují důležitou část jádra Linuxu. Stejně jako ostatní části operačního systému pracují v silně privilegovaném prostředí a pokud by něco dělaly špatně, mohly by způsobit závažné problémy. Ovladače zařízení řídí interakci mezi operačním systémem a tím hardwarovým zařízením, které ovládají. Například při zápisu na disk IDE používá souborový systém obecné rozhraní blokového zařízení. Ovladač se stará o detaily a zajišťuje provedení operací, specifických pro dané zařízení. Ovladače zařízení jsou určeny vždy pro konkrétní čip řadiče, s nímž spolupracují, takže pokud například máte SCSI řadič NCR810, budete potřebovat ovladač pro řadič NCR810.

2.2.4 Souborové systémy

V Linuxu, stejně jako v systému Unix, se k samostatným souborovým systémům, které systém může používat, nepřistupuje pomocí identifikátorů zařízení (jako jsou čísla nebo jména disků). Namísto toho jsou všechna zařízení zkombinována do jediné stromové hierarchické struktury, která reprezentuje souborový systém jako celek. Do tohoto jediného stromu přidává Linux nový souborový systém pokaždé, jakmile dojde k jeho připojení do připojovacího adresáře, řekněme `/mnt/cdrom`. Jednou z důležitých vlastností Linuxu je podpora řady roz-

dílných souborových systémů. Díky tomu je celý systém velmi pružný a je schopen spolupracovat s jinými operačními systémy. Nejoblíbenější souborový systém v Linuxu je systém `ext2`, který podporuje většina distribucí Linuxu.

Souborový systém dává uživateli rozumný pohled na soubory a adresáře na discích bez ohledu na typ souborového systému nebo jiné vlastnosti samotného fyzického zařízení. Linux transparentně podporuje řadu rozdílných souborových systémů (například *MS-DOS* a `ext2`) a všechny připojené soubory a souborové systémy představuje jako jediný integrovaný virtuální souborový systém. Obecně tedy platí, že uživatelé a procesy nepotřebují vědět typ souborového systému, v němž je uložen nějaký soubor, prostě jej jenom používají.

Blokové ovladače zařízení skrývají rozdíl mezi fyzickými typy blokových zařízení (například IDE a SCSI) a, z pohledu libovolného souborového systému, představují každé zařízení pouze jako lineární seznam bloků dat. Velikosti bloků se mohou pro různá zařízení lišit, například pro disketová zařízení je typický blok o velikosti 512 bajtů, pro zařízení IDE je typický blok o velikosti 1024 bajtů a tyto podrobnosti jsou před uživatelem systému skryty. Souborový systém `ext2` vypadá vždy stejně, bez ohledu na to, na jakém zařízení je fyzicky uložen.

2.3 Datové struktury jádra

Operační systém musí udržovat řadu informací o momentálním stavu systému. Jak se v systému různé věci mění, je nutné tyto datové struktury modifikovat tak, aby odrážely skutečnost. Když se například přihlásí uživatel, může se vytvořit nový proces. Jádro musí vytvořit datovou strukturu reprezentující nový proces a zapojit jej mezi datové struktury, reprezentující všechny ostatní procesy v systému.

Tyto datové struktury povětšinou existují pouze ve fyzické paměti a přistupovat k nim může pouze jádro a jeho subsystémy. Datové struktury obsahují data a ukazatele – adresy jiných datových struktur nebo rutin. Všechny dohromady mohou datové struktury používané jádrem Linuxu působit zmatečně. Každá datová struktura však má svůj účel a přestože některé jsou používány několika subsystémy jádra, jsou podstatně jednodušší, než by mohlo vypadat na první pohled.

Pochopení jádra Linuxu je vázáno pochopením jeho datových struktur a funkcí, k nimž je jádro Linuxu používá. Tato kniha zakládá popis jádra Linuxu na jeho datových strukturách. O každém subsystému jádra se zde hovoří v termínech jeho algoritmů, metod, jakými plní svou funkci, a způsobu použití datových struktur jádra.

2.3.1 Lineární seznamy

Při údržbě svých datových struktur používá Linux řadu technik softwarového inženýrství. V řadě příležitostí používá *propojené* nebo *zřetězené* datové struktury. Pokud každou datovou strukturu chápeme jako jednu instanci nebo výskyt něčeho, řekněme procesu nebo síťového zařízení, musí být jádro schopno nalézt všechny instance. V lineárním seznamu obsahuje kořenový ukazatel adresu první datové struktury (*prvku)*, v seznamu a každá struktura obsahuje ukazatel na následující prvek seznamu. Ukazatel posledního prvku má hodnotu 0 nebo NULL, čímž se signalizuje konec seznamu. V *obousměrně propojeném* seznamu obsahuje každý prvek ukazatel jednak na následující prvek a jednak také ukazatel na předchozí prvek seznamu. Pomocí obousměrně propojených seznamů se usnadňuje přidávání a odstraňování prvků uprostřed seznamu, je k tomu ale zapotřebí více paměti. To je typické dilema každého operačního systému: rozhodování mezi zatížením paměti a procesoru.

2.3.2 Hashovací tabulky

Lineární seznamy jsou užitečný nástroj jak organizovat datové struktury, průchod takovýmto seznamem ale může být neefektivní. Pokud hledáte určitý prvek, může se vám snadno stát, že budete muset procházet celým seznamem. K obejití tohoto omezení používá Linux další techniku, *hashování*. *Hashovací tabulka* je *pole* nebo *vektor* ukazatelů. Pole nebo vektor je jednoduše soustava věcí, které leží v paměti jedna za druhou. Knihovnu můžeme chápat jako pole knih. K polím se přistupuje pomocí *indexu*, který představuje offset v poli. Pokud se budeme dále držet analogie s knihovničkou, můžete každou knihu popsat její pozicí v knihovně, můžete například chtít pátou knihu.

Hashovací tabulka je pole ukazatelů na datové struktury, přičemž její index se odvozuje od informací v těchto strukturách. Pokud budete mít datovou strukturu obsahující informace o obyvatelích nějaké vesnice, můžete jako index použít věk obyvatel. Při hledání dat o určité osobě použijete její věk jako index do hashovací tabulky obyvatel a pak už budete pouze sledovat ukazatel na datovou strukturu obsahující informace o určité osobě. Bohužel je velmi pravděpodobné, že určitý věk bude mít více obyvatel ve vesnici, takže ukazatel v hashovací tabulce je ukazatelem na řetězec či seznam datových struktur, které popisují obyvatele stejného věku. Nicméně průchod těmito kratšími seznamy je stále rychlejší než prohledávání všech datových struktur.

Hashovací tabulky urychlují přístup k často používaným datovým strukturám, Linux používá hashovací tabulky velmi často při implementaci *vyrovnávacích pamětí*. Vyrovnávací paměť obsahuje užitečné informace, k nimž je nutné přistupovat rychle, a často obsahuje pouze podmnožinu všech dostupných dat. Datové struktury se ukládají a udržují ve vyrovnávacích pamětech, protože k nim jádro přistupuje velmi často. Vyrovnávací paměti mají ovšem i nevýhodu, protože jejich použití a údržba je mnohem složitější než prostá manipulace s lineár-

ními seznamy nebo hashovacími tabulkami. Pokud se nějakou datovou strukturu podaří ve vyrovnávací paměti najít (takzvaný *zásah*), udělá to radost. Pokud tam ale požadovaná struktura není, je nutné prohledat všechny příslušné struktury, a pokud požadovaná struktura existuje, musí se přidat do vyrovnávací paměti. Při přidávání nové struktury do vyrovnávací paměti může být nezbytné odstranit z ní nějakou jinou strukturu. Linux musí rozhodnout o tom, kterou strukturu odstranit, přičemž hrozí nebezpečí, že odstraněnou strukturu bude potřebovat hned vzápětí.

2.3.3 Abstraktní rozhraní

Jádro Linuxu používá velmi často abstraktních rozhraní. Rozhraní je soubor rutin a datových struktur, které nějakým způsobem fungují. Například všechny ovladače síťových zařízení musejí nabízet určité rutiny manipulující nad určitými datovými strukturami. Řešením mohou být obecné vrstvy kódu, které používají služeb (rozhraní) nižších vrstev. Síťová vrstva je obecná a ve spolupráci s kódem pro konkrétní zařízení vytváří standardní rozhraní.

Velmi často se nižší vrstvy při zavádění systému *registrují* u vyšších vrstev. Registrace obvykle obnáší přidání nějaké datové struktury do nějakého seznamu. Například každý souborový systém vestavěný v jádře se v jádru registruje při zavádění systému nebo, pokud používáte moduly, při prvním použití daného souborového systému. Které souborové systémy jsou registrovány můžete zjistit v souboru `/proc/filesystems`. Registrační datová struktura velmi často obsahuje ukazatele na jednotlivé funkce. Jedná se o adresy programových funkcí, které zajišťují určité úkony. Když použijeme opět jako příklad registraci souborového systému, v datové struktuře, kterou každý souborový systém předává jádru při své registraci, je obsažena adresa rutiny specifické pro daný souborový systém, která musí být zavolána vždy, když se systém připojuje.

Kapitola

3

Správa paměti

Subsystém pro správu paměti je jednou z nejdůležitějších částí operačního systému. Už od prvních dnů výpočetní techniky bylo vždy zapotřebí více paměti, než kolik v systému fyzicky existovalo. Byly vyvinuty různé strategie jak toto omezení obejít, jednou z nejúspěšnějších je použití virtuální paměti. Virtuální paměť způsobuje, že se systém tváří jako kdyby měl více paměti než kolik ve skutečnosti má díky tomu, že se paměť podle potřeby sdílí mezi jednotlivými procesy.

Virtuální paměť toho ovšem zajišťuje více než jenom zvětšení objemu skutečné paměti. Subsystém správy paměti zajišťuje následující funkce:

Velký adresový prostor

Operační systém způsobuje, že se systém chová, jako kdyby měl více paměti, než kolik ve skutečnosti má. Velikost virtuální paměti může být mnohonásobně větší než velikost skutečné fyzické paměti v systému.

Ochrana

Každý proces v systému má vlastní virtuální adresový prostor. Virtuální adresové prostory jsou vzájemně úplně odděleny a běžící proces jedné aplikace tak nemůže nijak ovlivnit jiné procesy. Navíc mohou hardwarové mechanismy virtuální paměti chránit určité oblasti paměti před zápisem. Tím se kód a data chrání před přepsáním chybnými aplikacemi.

Mapování paměti

Mapování paměti slouží k mapování obrazů programu a datových souborů do adresového prostoru procesu. Při použití paměťového mapování je obsah souboru přímo svázán s virtuálním adresovým prostorem procesu.

Alokace fyzické paměti

Subsystém správy paměti umožňuje každému spuštěnému procesu alokovat spravedlivý díl fyzické paměti systému.

Sdílení virtuální paměti

Přestože virtuální paměť umožňuje, aby procesy měly oddělené (virtuální) adresové prostory, jsou situace, kdy je vhodné sdílet paměť mezi více procesy. V systému může být například více procesů, které provádějí příkaz `bash`. Není nutné mít v paměti více kopií programu `bash`, každý v adresovém prostoru jednoho procesu, je lepší mít ve fyzické paměti pouze jedinou kopii, kterou budou příslušné procesy sdílet. Dalším běžným příkladem sdílení kódu mezi více procesy jsou například dynamické knihovny.

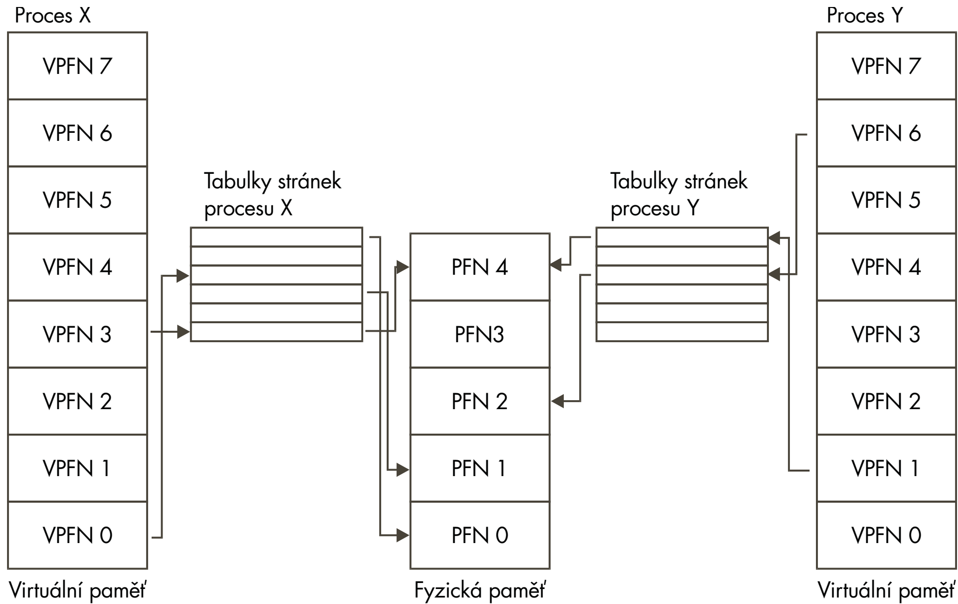
Sdílenou paměť je možno využít také jako mechanismus pro meziprocesovou komunikaci, kdy si dva nebo více procesů vyměňují informace prostřednictvím paměťové oblasti, kterou společně sdílejí. Linux podporuje meziprocesovou komunikaci sdílením paměti mechanismem Unix System V IPC.

3.1 Abstraktní model virtuální paměti

Před popisem metod, které Linux používá pro podporu virtuální paměti, bude užitečné seznámit se s abstraktním modelem, který nebude zatížen přílišnými detaily.

Když procesor provádí program, načítá z paměti instrukce a dekóduje je. Při dekódování instrukcí může potřebovat načíst nebo zapsat z nebo na určité místo paměti. Pak procesor instrukci provede a přesune se na následující instrukci programu. Procesor tedy neustále přistupuje k paměti, ať už načítáním instrukce nebo načítáním či zápisem dat.

Při použití virtuální paměti jsou všechny adresy virtuální, nejsou to adresy fyzické. Virtuální adresy překládá na fyzické adresy procesor podle informací v tabulkách, udržovaných operačním systémem.



Obrázek 3.1

Abstraktní model mapování virtuálních adres na fyzické adresy

Aby se překlad zjednodušil, je virtuální i fyzická paměť rozdělena na pohodlně manipulovatelné části, takzvané *stránky*. Všechny stránky jsou stejně veliké. Teoreticky by sice nemusely být, ale kdyby nebyly, celý systém by se jen velmi obtížně spravoval. Linux na systémech s procesorem Alpha AXP používá stránky o velikosti 8 KB, systémy s procesorem Intel *x*86 používají stránky o velikosti 4 KB. Každá stránka je označena jedinečným číslem, číslem rámce stránky (PFN).

V takovémto stránkovém modelu se virtuální adresa skládá ze dvou částí: offsetu a čísla rámce virtuální stránky. Vezmeme-li stránku o velikosti 4 KB, bity 11 až 0 virtuální adresy jsou offset a bity 12 a výše představují číslo rámce stránky. Procesor musí přeložit číslo rámce virtuální stránky na fyzickou a poté ve fyzické stránce přistoupit na místo se správným offsetem. K této činnosti procesor používá *tabulky stránek*.

Na obrázku 3.1 vidíme virtuální adresový prostor dvou procesů, procesu *X* a procesu *Y*, každý má svou vlastní tabulku stránek. Tyto tabulky mapují každou virtuální stránku procesu na fyzickou stránku paměti. Vidíme, že virtuální stránka 0 procesu *X* je mapována na fyzickou stránku 1, virtuální stránka 1 procesu *Y* se mapuje na fyzickou stránku 4. Každá položka v naší pomyslné tabulce stránek musí obsahovat následující informace:

- Příznak platnosti, který udává, zda je položka tabulky platná.
- Číslo rámce fyzické stránky, kterou tato položka popisuje.
- Informace pro řízení přístupu, které popisují, jak se stránka může používat. Smí se do ní zapisovat? Obsahuje spustitelný kód?

K tabulce stránek se přistupuje tak, že číslo rámce virtuální stránky slouží jako offset v tabulce stránek. Virtuální stránka 5 tedy bude šestou položkou tabulky (první položkou je stránka 0).

Při překladu virtuální adresy na fyzickou musí procesor nejprve zjistit číslo rámce virtuální stránky a poté offset v tabulce virtuálních stránek. Když je velikost stránky mocninou dvou, dají se tyto operace snadno provádět pomocí bitového maskování a posuvů. Podívejme se znovu na obrázek 3.1 a předpokládejme stránku o velikosti *0x2000* bajtů (desítkově 8192) a adresu *0x2194* ve virtuálním adresovém prostoru procesu *Y*, kterou procesor přeloží jako adresu na offsetu *0x194* ve virtuální stránce 1.

Číslo rámce virtuální stránky procesor použije jako index do tabulky stránek a získá tak položku tabulky pro danou stránku. Pokud je položka tabulky s daným offsetem platná, procesor z ní převezme číslo rámce fyzické paměti. Pokud položka není platná, proces se pokouší o přístup k neexistující oblasti virtuální paměti. V takovém případě procesor nemůže provést překlad adresy a musí předat řízení operačnímu systému, který problém vyřeší.

Jak ale procesor doručí operačnímu systému zprávu o tom, že se nějaký proces pokusil o přístup k oblasti virtuální paměti, kterou procesor nemohl přeložit na fyzickou adresu? Ať už je mechanismus předání zprávy jakýkoliv, této události se obecně říká *výpadek stránky* a operační systém je upozorněn na výpadek virtuální paměti a na okolnosti, za kterých k tomu došlo.

Pokud se proces odkazuje na platnou položku tabulky stránek, procesor vezme číslo fyzického rámce stránky a vynásobí jej velikostí stránky, takže získává *bázovou* adresu stránky ve fyzické paměti. K této adrese se přičte offset instrukce nebo dat, která se požadují.

Když se budeme držet našeho předchozího příkladu, mapuje se virtuální stránka 1 procesu *Y* na fyzickou stránku 4, která začíná na adrese *0x8000* (4 × *0x2000*). Přičtením bajtového offsetu *0x194* se dostáváme na fyzickou adresu *0x8194*.

Tento mechanismus mapování virtuálních adres na fyzické umožňuje, aby se stránky virtuální paměti mapovaly na fyzické stránky v libovolném pořadí. Například na obrázku 3.1 se virtuální stránka 0 procesu *X* mapuje na fyzickou stránku 1, zatímco virtuální stránka 7 se mapuje na fyzickou stránku 0, přestože se jedná o vyšší virtuální adresu, než je adresa stránky 0. To nám ukazuje zajímavý vedlejší efekt virtuální paměti - stránky virtuální paměti nemusejí být ve fyzické paměti přítomny v žádném pevném pořadí.

3.1.1 Vynucené stránkování

Protože fyzické paměti je v systému méně než paměti virtuální, musí být operační systém velmi opatrný na to, aby neplýtval fyzickou pamětí. Jedna možnost jak ušetřit fyzickou paměť je nahrávat pouze ty virtuální stránky, které právě spuštěný program opravdu potřebuje. Mějme například spuštěn nějaký databázový program, který něco hledá v databázi. V takovém případě není nutné, aby byla do paměti nahrána celá databáze, stačí nahrát pouze ty záznamy, které se momentálně zkoumají. Pokud databázi prohledáváme, není nutné mít v paměti zároveň nahránu tu část programu, která zajišťuje přidávání nových záznamů. Této metodě nahrávání pouze potřebných stránek se říká vynucené stránkování.

Když se proces pokusí o přístup k virtuální adrese, která momentálně není v paměti, procesoru se nepodaří pro požadovanou stránku najít platný záznam v tabulce stránek. Například podle obrázku 3.1 není v tabulce stránek procesu *X* platná položka pro virtuální stránku 2, takže pokud by se proces *X* pokusil o čtení virtuální adresy ve druhé stránce, procesoru by se nepodařilo provést překlad na fyzickou adresu. V této chvíli procesor upozorní operační systém na fakt, že došlo k výpadku stránky.

Pokud je virtuální adresa neplatná, znamená to, že aplikace se pokusila o přístup k adrese, k níž by přistupovat neměla. Aplikace třeba někam zabloudila a pokouší se nyní zapisovat na nějakou náhodnou adresu. V takovém případě ji operační systém ukončí, čímž chrání ostatní procesy v systému před chybnou aplikací.

Pokud je ale adresa platná a odkazovaná stránka momentálně není v paměti, musí operační systém přesunout tuto stránku do paměti z obrazu paměti uloženého na disku. Diskové přístupy trvají relativně dlouho, takže proces bude muset počkat, než dojde k načtení stránky. Pokud jsou v systému současně jiné procesy, které běžet mohou, nechá operační systém mezitím pracovat tyto procesy. Načtená stránka se zapíše do některé volné stránky fyzické paměti a do tabulky stránek procesu se doplní záznam ke dříve požadované virtuální stránce. Proces se pak znovu spustí na stejné instrukci, která vyvolala výpadek stránky. Tentokrát se při přístupu do virtuální paměti procesoru podaří přeložit virtuální adresu na fyzickou a proces tedy může pokračovat v práci.

Linux používá vynucené stránkování při nahrávání obrazů spustitelných souborů do virtuální paměti procesů. Když je spuštěn nějaký příkaz, otevře se soubor, který jej obsahuje, a jeho obsah se namapuje do virtuální paměti procesu. Dosáhne se toho modifikací datových struktur jádra, které popisují paměťovou mapu procesu a celý postup se označuje jako *mapování paměti*. Nicméně do fyzické paměti se nahraje pouze první část obrazu programu. Zbytek zůstává na disku. Když se obraz provádí, generuje výpadky stránek a Linux pomocí paměťové mapy procesu zjišťuje, které části obrazu je potřeba nyní přenést do paměti a nechat je proběhnout.

3.1.2 Odkládání na disk

Když proces potřebuje přenést do fyzické paměti nějakou stránku a ve fyzické paměti není žádná stránka volná, musí operační systém pro požadovanou stránku vytvořit místo tím, že ve fyzické paměti zruší nějakou jinou stránku.

Pokud rušená stránka pochází z obrazu spustitelného souboru nebo z datového souboru do nějž nebylo zapisováno, není nutné stránku ukládat. Je možno ji přímo zrušit a pokud by ji příslušný proces potřeboval znovu, načetla by se zpátky do paměti z příslušného obrazu nebo datového souboru.

Pokud ale stránka byla modifikována, musí operační systém zachovat její obsah tak, aby k ní bylo možno později opět přistupovat. Tomuto typu stránek se říká *modifikované* (*dirty*) stránky a při jejich odstraňování z paměti se ukládají do speciálního souboru, zvaného odkládací soubor. Přístupy k odkládacímu souboru jsou výrazně pomalejší než přístupy k fyzické paměti a operační systém tedy musí pečlivě balancovat mezi potřebami zapisovat stránky na disk a potřebami znovu je načítat při dalším použití.

Pokud je algoritmus používaný k rozhodnutí o tom, které stránky zrušit či odkládat (takzvaný *odkládací algoritmus*) neefektivní, dochází k situaci zvané *thrashing* - stránky se neustále zapisují na disk a čtou z disku a operační systém je vytížen natolik, že nemá čas na pořádnou práci. Pokud by například bylo často přistupováno k fyzické stránce 1 na obrázku 3.1, pak tato stránka není vhodným kandidátem pro odložení na disk. Množina stránek, které proces momentálně používá, se nazývá *pracovní prostor*. Efektivní odkládací mechanismus musí zajistit, aby pracovní prostor každého procesu byl přítomen ve fyzické paměti.

Linux používá k volbě stránek, které se mají z paměti odsunout, mechanismus zvaný LRU - Least Recently Used, tedy česky „nejdávněji použitá“. Podle tohoto schématu má každá stránka v systému svůj „věk“) který se mění s přístupem ke stránce. Čím častěji se ke stránce přistupuje, tím je „mladší“) méně navštěvovaná stránka je starší a stále stárne. Dobrým kandidátem na odložení jsou právě staré stránky.

3.1.3 Sdílená virtuální paměť

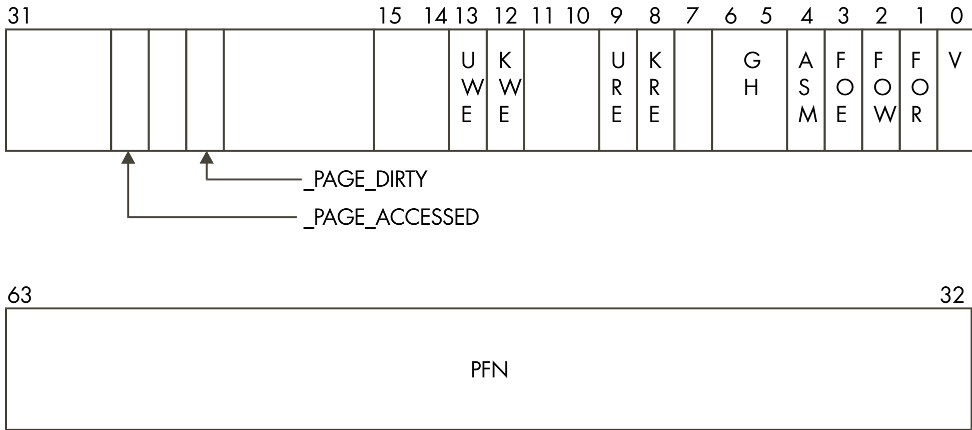
Virtuální paměť zjednodušuje sdílení paměti mezi několika procesy. Všechny přístupy do paměti se provádějí přes tabulky stránek a každý proces má svou vlastní tabulku stránek. Pokud mají dva procesy sdílet stejný kus fyzické paměti, objeví se v tabulkách stránek obou procesů stejné číslo fyzického paměťového rámce.

Na obrázku 3.1 vidíme dva procesy, které sdílejí fyzickou stránku 4. Pro proces *X* je to virtuální stránka 4, zatímco pro proces *Y* je to virtuální stránka 6. Tím se ukazuje zajímavý rys sdílení stránek: sdílené fyzické stránky se nemusejí ve virtuálním adresovém prostoru procesů, které je sdílejí, nacházet na stejných místech.

3.1.4 Fyzické a virtuální adresovací režimy

Není příliš rozumné, aby samotný operační systém běžel ve virtuální paměti. Bylo by dost příšerné, kdyby operační systém musel udržovat tabulku stránek i sám pro sebe. Většina univerzálních procesorů podporuje jak fyzický adresovací režim, tak virtuální adresovací režim. Fyzický adresovací režim nepotřebuje žádné tabulky stránek a v tomto režimu procesor neprovádí žádné překlady adres. Jádro Linuxu je sestaveno tak, aby pracovalo ve fyzickém adresovacím režimu.

Procesor Alpha AXP nemá žádný zvláštní fyzický adresovací režim. Namísto toho rozděluje paměťový prostor na několik oblastí a dvě z nich určuje jako oblasti s fyzicky mapovanými adresami. Tento adresový prostor jádra se označuje jako segment KSEG a zahrnuje všechny adresy od *0xfffffc0000000000* nahoru. Aby bylo možno provádět kód v segmentu KSEG (podle definice kód jádra) a přistupovat k datům v této oblasti, musí kód běžet v režimu jádra. Jádro Linuxu pro procesory Alpha je sestaveno tak, aby se spouštělo od adresy *0xfffffc0000310000*.



Obrázek 3.2
Položka tabulky stránek procesoru Alpha AXP

3.1.5 Řízení přístupu

Záznamy v tabulkách stránek obsahují také informace o řízení přístupu. Protože procesor tyto tabulky využívá pro mapování virtuálních adres procesů na fyzické, může při té příležitosti použít i informace o řízení přístupu ke kontrole, zda proces nepřistupuje k paměti nepovoleným způsobem.

Existuje řada důvodů proč omezovat přístup do různých oblastí paměti. Některé části paměti, například ty, kde je uložen spustitelný kód, jsou přirozeně určeny pouze pro čtení, operační systém by neměl procesu dovolit přepisovat data přes svůj spustitelný kód. Naopak stránky obsahující data je možno modifikovat, neměl by se však podařit pokus o provedení obsahu těchto stránek jako programu. Většina procesorů má nejméně dva režimy běhu programů: režim *jádra* a *uživatelský* režim. Zřejmě nebudete chtít, aby byl kód jádra a jeho datové struktury přístupné, pokud proces neběží v režimu jádra.

Informace pro řízení přístupu jsou udržovány v PTE (položkách tabulek stránek) a jsou procesorově závislé. Na obrázku 3.2 vidíme strukturu PTE procesoru Alpha AXP. Jednotlivé bity mají následující význam:

V „Valid“; je nastaven, pokud je PTE platná.

FOE „Fault on Execute“; když dojde k pokusu o provádění instrukcí v této stránce, ohlásí procesor výpadek stránky a předá řízení operačnímu systému.

FOW „Fault on Write“; jako výše, k výpadku však dochází při pokusu o zápis do této stránky.

FOR „Fault on Read“; jako výše, k výpadku však dochází při pokusu o čtení z této stránky.

ASM „Address Space Match“; tento příznak se používá pokud chce operační systém vyčistit pouze některé položky z překladového bufferu.

KRE Kód běžící v režimu jádra může tuto stránku číst.

URE Kód běžící v uživatelském režimu může tuto stránku číst.

GH Příznak granularity slouží při mapování celého bloku jedním překladovým bufferem a ne několika.

KWE Kód běžící v režimu jádra může do této stránky zapisovat.

UWE Kód běžící v uživatelském režimu může do této stránky zapisovat.

PFN Číslo rámce stránky. U PTE s nastaveným příznakem V obsahuje toto pole číslo fyzického rámce stránky. Pokud příznak V není nastaven a toto pole není nulové, obsahuje informace o tom, kde je stránka uložena ve odkládacím souboru.

Následující dva bity jsou definovány a využívány Linuxem:

_PAGE_DIRTY Pokud je příznak nastaven, stránku je nutno uložit do odkládacího souboru.

_PAGE_ACCESSED Tímto příznakem Linux označuje stránku, k níž se přistupovalo.

3.2 Vyrovnávací paměti

Pokud byste chtěli systém implementovat podle právě popsaného teoretického modelu, systém by sice mohl fungovat, nepracoval by ale příliš efektivně. Návrháři operačních systémů i procesorů usilují o co největší zvýšení výkonu systému. Kromě zrychlení samotného procesoru, paměti a dalších prvků je nejlepší řešení použití vyrovnávacích pamětí pro užitečné informace a data, které zajistí rychlejší provádění některých operací. Při správě paměti využívá Linux řadu vyrovnávacích pamětí.

Vyrovnávací paměť bufferů

Vyrovnávací paměť bufferů obsahuje datové buffery, které využívají ovladače blokových zařízení. Tyto buffery mají pevnou délku (například 512 bajtů) a obsahují bloky informací, které už byly z blokového zařízení načteny, nebo které se na něj mají zapsat. S blokovými zařízeními se dá pracovat pouze tím způsobem, že se z nich přečtou nebo se na ně zapíší bloky pevné délky. Všechny pevné disky jsou bloková zařízení.

Buffery jsou indexovány identifikátorem zařízení a číslem požadovaného bloku a slouží k rychlému nalezení bloku dat. K blokovým zařízením se vždy přistupuje pouze přes buffery. Pokud se data nacházejí v bufferu, není nutné je načíst z fyzického blokového zařízení (řekněme pevného disku) a přístup se tak výrazně zrychluje.

Vyrovnávací paměť stránek

2 Slouží ke zrychlení přístupu k obrazům programů a k datům na disku.

Ukládá se do ní logický obsah souborů stránek a přistupuje se k ní prostřednictvím identifikace souboru a offsetu. Při načítání stránek z disku do paměti se stránky uchovávají v této vyrovnávací paměti.

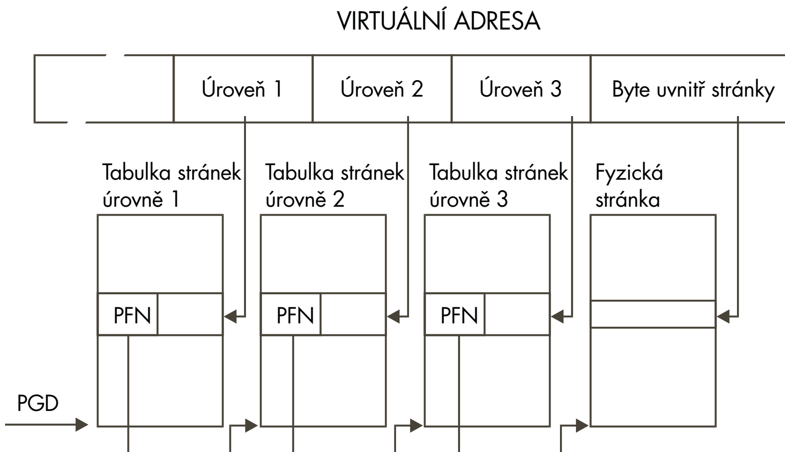
Odkládací paměť

3 V odkládacím souboru se ukládají pouze modifikované stránky.

Pokud nedojde k modifikaci stránky od posledního zápisu do odkládacího souboru, pak při následující potřebě stránku odkládat není nutné ji do souboru zapisovat, protože už v něm zapsána je. Namísto toho je možno stránku pouze zrušit. V silně odkládacím systému se tak ušetří zbytečné a náročné diskové operace.

Hardwarové vyrovnávací paměti

Jednou běžně implementovanou hardwarovou vyrovnávací pamětí je přímo v procesoru vestavěná vyrovnávací paměť položek tabulek stránek. V takovém případě procesor nemusí pokaždé přímo načítat položku tabulky stránek, podle potřeb využívá informace uložené ve své vyrovnávací paměti. Tato paměť se označuje jako překladový buffer (Translation Look-aside Buffer, TLB) a obsahuje kopie položek tabulek stránek jednoho nebo více procesů v systému.



Obrázek 3.3
Tříúrovňové tabulky stránek

Když dojde k odkazu na virtuální adresu, procesor se pokusí najít potřebnou položku TLB. Pokud ji najde, může virtuální adresu přímo přeložit na fyzickou adresu a provést požadovanou datovou operaci. Pokud se procesoru nepodaří najít potřebnou položku TLB, musí požádat o pomoc operační systém. Provede to tak, že signalizuje výpadek bloku TLB. Systémově závislé mechanismy zajistí doručení této výjimky operačnímu systému, který může problém napravit. Operační systém vygeneruje novou položku TLB pro požadované mapování. Po ošetření výjimky se procesor opět pokusí o překlad virtuální adresy. Tentokrát už všechno bude fungovat, protože pro požadovanou adresu existuje potřebný blok TLB.

Nevýhodou použití vyrovnávacích pamětí, ať už hardwarových nebo jiných, je, že kvůli ušetření práce musí Linux věnovat více času a prostoru údržbě vyrovnávacích pamětí a pokud by došlo k porušení obsahu vyrovnávacích pamětí, celý systém se zhroutí.

3.3 Tabulky stránek Linuxu

Linux předpokládá tři úrovně tabulky stránek. Každá tabulka stránek obsahuje číslo rámce stránky tabulky stránek další úrovně. Na obrázku 3.3 je vidět, jak se virtuální adresa rozpadá na řadu položek, každá z nich představuje offset v určité tabulce stránek. Při překladu virtuální adresy na fyzickou musí procesor vzít obsah položky na každé úrovni, konvertovat jej na offset ve fyzické paměti obsahující tabulku stránek a načíst číslo rámce stránky další úrovně tabulky stránek. Toto se opakuje třikrát až dojde k nalezení čísla rámce fyzické stránky, která obsahuje požadovanou virtuální adresu. V tom okamžiku se použije poslední pole virtuální adresy, bajtový offset, kterým se naleznou požadovaná data ve stránce.

Každá platforma, na níž Linux běží, musí zajistit překladová makra pro průchod tabulkou stránek procesu. Díky tomu jádro nemusí znát formát položek tabulky stránek ani jejich uspořádání.

Toto řešení je natolik úspěšné, že Linux může používat stejný kód pro manipulaci s tabulkami stránek pro procesor Alpha, který pracuje se třemi úrovněmi tabulky stránek, i pro procesor Intel *x*86, který používá pouze dvouúrovňovou tabulku.

3.4 Alokace a dealokace stránek

V systému se vyskytuje velká řada požadavků na fyzické stránky. Když například dochází k nahrávání obrazu kódu do paměti, systém potřebuje alokovat stránky. Tyto stránky budou uvolněny až kód skončí svou práci. Další použití fyzických stránek je pro uložení datových struktur jádra, jako například samotných tabulek stránek. Mechanismy a datové struktury používané pro alokování a dealokování stránek jsou pravděpodobně nejkritičtější pro efektivnost celého subsystému virtuální paměti.

5 Všechny fyzické stránky jsou popsány datovou strukturou mem_map, což je seznam struktur mem_map_t [1], které se inicializují při zavádění systému. Každá struktura mem_map_t popisuje jednu fyzickou stránku v systému. Důležitými položkami struktury (co se týče správy paměti) jsou následující položky:

count Počitadlo počtu uživatelů této stránky. Pokud je hodnota počitadla větší než jedna, je stránka sdílena mezi více procesy.

age Tato položka popisuje věk stránky a slouží k rozhodování, zda je stránka vhodným kandidátem na zrušení a odložení.

map_nr Číslo rámce fyzické stránky, kterou tato struktura mem_map_t popisuje.

Vektor free_area slouží kódu alokace stránek k nalezení volných stránek. Tímto mechanismem je podporováno celé schéma správy bufferů a co se týče samotného kódu, velikost stránky a fyzický stránkovací mechanismus konkrétního procesoru jsou irelevantní.

Každý prvek struktury free_area obsahuje informace o bloku volných stránek. První prvek pole popisuje samostatné stránky, druhý prvek bloky o dvou stránkách, následující prvek bloky o čtyřech stránkách a tak dále s krokem po mocninách dvou. Položka list se používá jako začátek fronty a obsahuje ukazatel na datové struktury page v poli mem_map. Zde se řadí do fronty bloky stránek. map je ukazatel na bitovou mapu, která obsahuje záznamy o alokovaných skupinách stránek dané velikosti. N-tý bit této bitové mapy je nastaven, pokud je N-tý blok stránek volný.

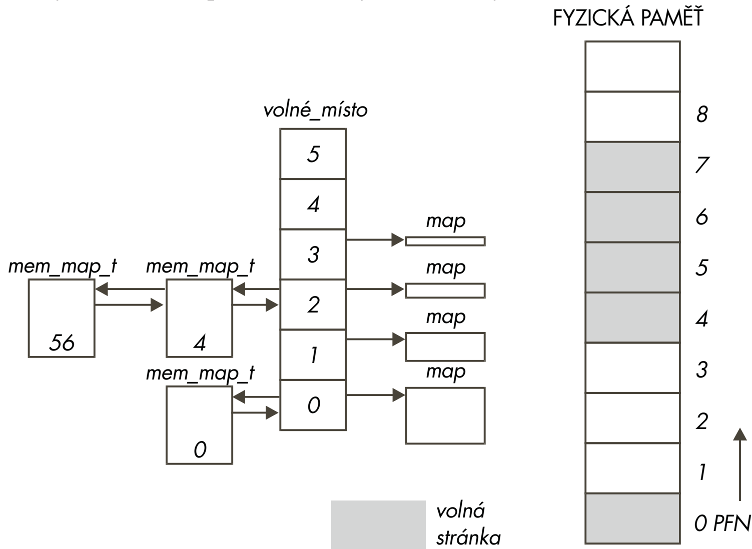
Na obrázku 3.4 vidíme strukturu free_area. Prvek 0 má jednu volnou stránku (stránku s číslem rámce 0) a prvek 2 má dva volné bloky o čtyřech stránkách, první začíná stránkou s číslem rámce 4 a druhý stránkou s číslem rámce 56.

3.4.1 Alokace stránek

6 K efektivní alokaci a dealokaci bloků stránek používá Linux Buddyho algoritmus. Kód pro alokování stránek se pokouší alokovat blok jedné nebo více fyzických stránek. Stránky se alokují v blocích o velikosti mocniny dvou. Znamená to tedy, že můžete alokovat blok o jedné stránce, dvou stránkách, čtyřech stránkách a tak dále. Dokud je v systému dostatek volných stránek, aby mohl být požadavek splněn (nr_free_pages > min_free_pages), bude alokační kód prohledávat strukturu free_area a bude hledat blok o požadovaném počtu stránek. Každý prvek seznamu free_area je mapa alokovaných a volných bloků stránek dané velikosti. Například prvek 2 pole je paměťová mapa, která popisuje volné a alokované bloky o velikosti 4 stránky.

[1] Je poněkud matoucí, že tato struktura se označuje také jako *stránka*.

Alokační algoritmus nejprve hledá bloky stránek požadované velikosti. Prochází seznamem volých stránek který je seřazen v prvku `list` datové struktury `free_area`. Pokud není nalezen žádný volný blok požadované velikosti, hledá se volný blok o větší velikosti (tedy blok dvakrát větší než požadovaný). Tento proces pokračuje do doby, než je prohledána celá struktura `free_area` nebo než bude nalezen vhodný blok stránek. Pokud je nalezen blok stránek o větší než požadované velikosti, musí se rozdělit na blok požadované velikosti. Protože velikosti bloků jsou vždy mocninami dvou, dělení bloků na menší části je prostě jenom jejich půlení.Volné bloky se zařadí do příslušné fronty a alokovaný blok stránek se vrátí volajícímu procesu.



Obrázek 3.4
Datová struktura `free_area`

Pokud si vezmeme například obrázek 3.4 a budeme požadovat blok o velikosti dvě stránky, dojde k rozdělení prvního bloku o čtyřech stránkách (který začíná stránkou s číslem rámce 4) na dva dvoustránkové bloky. První z nich, začínající na stránce s rámcem 4, bude vrácen volajícímu procesu jako požadovaný blok, druhý blok, začínající stránkou s číslem rámce 6, bude zařazen do fronty volných dvoustránkových bloků v prvku 1 pole `free_area`.

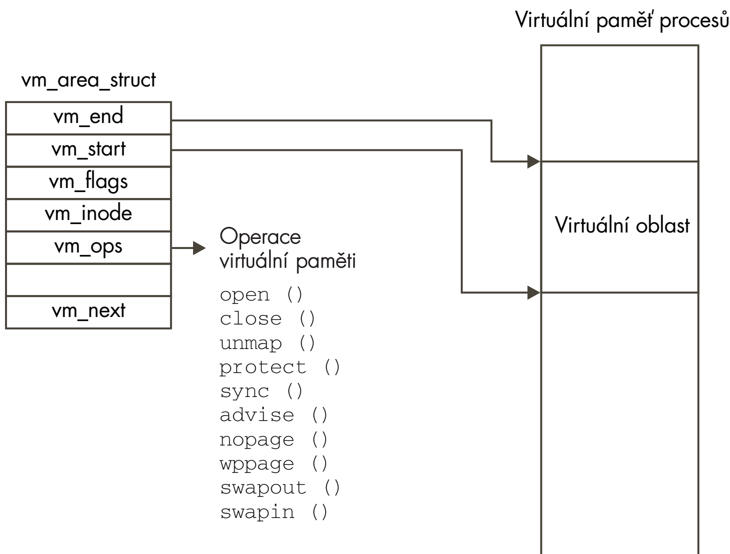
3.4.2 Dealokace stránek

Alokování bloků stránek vede k fragmentaci paměti, kdy se větší bloky volných stránek rozbíjejí na menší. Dealokační kód rekombinuje stránky zpět na větší bloky vždy když je to možné. Velikost vraceného bloku stránek je významná, protože umožňuje snadnou kombinaci bloků na větší bloky.

Vždy když dojde k uvolnění bloku stránek, kontroluje se, zda není volný sousední blok stránek o stejné velikosti. Pokud ano, pak se oba dva bloky zkombinují do nového volného bloku o dvojnásobné velikosti. Pokaždé když dojde k rekombinaci dvou bloků do většího bloku, pokusí se dealokační kód rekombinovat tento blok do ještě většího bloku. Díky tomu budou bloky volných stránek vždy tak velké, jak to jen využití paměti dovolí. Například pokud by na obrázku 3.4 došlo k uvolnění stránky s číslem rámce 1, zkombinovala by se s už volnou stránkou s číslem rámce 0 a výsledný blok by byl zařazen do prvního prvku struktury `free_area` jako blok o velikosti 2 stránky.

3.5 Mapování paměti

Když dochází k vykonávání obrazu programu, musí se obraz spustitelného kódu přenést do virtuálního adresového prostoru procesu. To platí i pro všechny sdílené knihovny, které jsou ke spustitelnému obrazu přilinkovány. Spustitelný soubor se fakticky nepřenáší do fyzické paměti, dojde pouze k jeho napojení na virtuální paměť procesu. Pak, když se spuštěná aplikace odkazuje na danou část programu, dojde k přenesení spustitelného obrazu do paměti. Toto napojení obrazu na virtuální adresový prostor procesu se označuje jako mapování paměti.



Obrázek 3.5
Oblasti virtuální paměti

Virtuální paměť každého procesu je reprezentována datovou strukturou `mm_struct`. Ta obsahuje informace o právě prováděném obrazu (například *bash*) a dále ukazatele na řadu datových struktur `vm_area_struct`. Každá datová struktura `vm_area_struct` popisuje začátek a konec oblasti virtuální paměti, přístupová práva procesu do této oblasti a operace pro tuto oblast paměti. Tyto operace jsou vlastně rutiny, které Linux musí používat při manipulaci s touto

oblastí virtuální paměti. Například jednou z operací nad oblastí virtuální paměti je provedení správné akce v případě, kdy se proces pokusí o přístup do této oblasti, ale zjistí (prostřednictvím výpadku stránky), že požadovaná oblast není momentálně ve fyzické paměti. Této operaci se říká operace *nopage*. Operace *nopage* se používá když Linux potřebuje vynutit přítomnost stránky spustitelného obrazu v paměti.

Když se spustitelný obraz mapuje do virtuální paměti procesu, generují se datové struktury `vm_area_struct`. Každá struktura `vm_area_struct` představuje část spustitelného obrazu: spustitelný kód, inicializovaná data (proměnné), neinicializovaná data a další. Linux podporuje řadu standardních operací nad virtuální pamětí a když dochází k vytváření struktury `vm_area_struct`, přiřadí se jí správná množina operací nad virtuální pamětí.

3.6 Vynucené stránkování

Jakmile je spustitelný obraz namapován do virtuální paměti procesu, může se začít provádět. Protože je v paměti fyzicky přítomen pouze úplný začátek spustitelného obrazu, dojde záhy k přístupu do oblasti virtuální paměti, která ještě není umístěna ve fyzické paměti. Když se proces pokouší o přístup k virtuální adrese, která nemá platnou položku tabulky stránek, procesor ohlásí Linuxu výpadek stránky.

Výpadek stránky oznamuje virtuální adresu, na níž došlo k výpadku, i typ paměťové operace, která výpadek způsobila.

Linux musí nalézt strukturu `vm_area_struct` té oblasti paměti, v níž došlo k výpadku stránky. Prohledávání struktur `vm_area_struct` je kritické pro efektivní obsluhu výpadků stránek, proto jsou struktury propojeny do takzvaného stromu AVL (Adělson-Velski a Landis) stromu. Pokud pro vypadlou virtuální adresu neexistuje datová struktura `vm_area_struct`, pokusil se proces o přístup na nelegální virtuální adresu. Linux odešle procesu signál `SIGSEGV` a pokud proces tento signál neobslouží, bude ukončen.

Dále Linux kontroluje typ vzniklého výpadku stránky proti typům přístupu povoleným v této oblasti virtuální paměti. Pokud se proces pokouší přistupovat k paměti nepovoleným způsobem, řekněme že chce zapisovat někam, kde je povoleno pouze čtení, bude mu rovněž signalizována paměťová chyba.

Když Linux zjistí, že výpadek stránky je legální, musí jej obsloužit.

Linux musí rozlišovat mezi stránkami uloženými v odkládacím souboru a mezi stránkami, které jsou součástí spustitelného obrazu někde na disku. Zjistí to podle položky tabulky stránek pro vypadlou stránku.

Pokud je položka tabulky stránek neplatná, ale neprázdná, znamená to, že vypadlá stránka je momentálně držena v odkládacím souboru. U položek tabulky stránek procesoru Alpha AXP jsou to ty položky, které nemají nastaven příznak platnosti, v poli PFN však obsahují nenulovou hodnotu. V tomto případě obsahuje PFN informaci o tom, kde v odkládacím souboru (a ve kterém odkládacím souboru) je stránka uložena. Jak jsou obsluhovány stránky uložené v odkládacím souboru bude popsáno dále v této kapitole.

Ne všechny datové struktury `vm_area_struct` mají přiděleny operace nad virtuální pamětí, a dokonce i ty, které je přiděleny mají, nemusejí mít přidělenu operaci *nopage*. Je to dáno tím, že implicitně Linux řeší přístup alokováním nové fyzické stránky a vytvořením platné položky tabulky stránek pro tuto stránku. Pokud není pro danou oblast paměti definována operace *nopage*, provede Linux implicitní obsluhu.

10

Obecná operace *nopage* se používá pro paměťově mapované spustitelné obrazy a používá vyrovnávací paměť stránek k přenesení požadovaného obrazu stránky do fyzické paměti.

Ať už dojde k přenosu stránky do fyzické paměti jakýmkoliv způsobem, provede se aktualizace tabulky stránek procesu. Při aktualizaci tabulky může být potřebné provést i nějaké hardwarově závislé operace, zejména pokud procesor používá interní překladové buffery. Nyní tedy byl výpadek stránky obsloužen, dále se o něj nestaráme a spouštíme proces znovu od instrukce, která výpadek stránky vyvolala.

3.7 Vyrovnávací paměť stránek v Linuxu

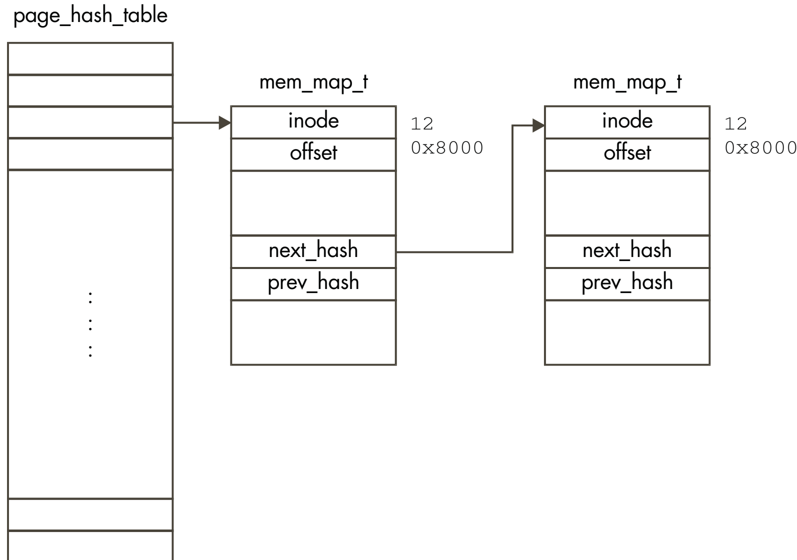
11

Úkolem vyrovnávací paměti stránek je zrychlit přístup k souborům na disku. Paměťově mapované soubory se načítají po jednotlivých stránkách a ty se ukládají do vyrovnávací paměti stránek. Na obrázku 3.6 vidíme, že vyrovnávací paměť stránek se skládá ze struktury `page_hash_table`, vektoru ukazatelů na datové struktury `mem_map_t`.

Každý soubor v Linuxu je identifikován datovou strukturou *inode* subsystému VFS (popsaného v kapitole Souborový systém). Každý inode je jedinečné číslo a plně odpovídá jednomu a právě jednomu souboru. Index do tabulky stránek se odvozuje od *inode* souboru a offsetu v tomto souboru.

Vždy, když se z paměťově mapovaného souboru načítá stránka, například pokud vznikl požadavek na přenesení do paměti v důsledku vynuceného stránkování, načítá se stránka přes vyrovnávací paměť stránek. Pokud je stránka ve vyrovnávací paměti přítomna, vrací se kódu obsluhy výpadku stránky ukazatel na strukturu `mem_map_t`, která odpovídá požadované stránce. V opačném případě se musí stránka přenést do paměti ze souborového systému. Linux provede alokaci fyzické stránky a načte stránku ze souboru na disku.

Pokud je to možné, zahájí Linux rovněž načtení následující stránky ze souboru. Toto jednoduché dopředné čtení jedné stránky znamená, že pokud proces přistupuje ke stránkám v lineárním pořadí (což je pravděpodobné), bude mít v okamžiku požadavku na další stránku tuto stránku už připravenu.



Obrázek 3.6
Vyrovnávací paměť stránek v Linuxu

Postupem času se obsah vyrovnávací paměti stránek rozrůstá, tak jak dochází k načítání a vykonávání obrazů. Stránky se z vyrovnávací paměti odstraňují v okamžiku, kdy nejsou déle potřebné, tedy pokud není daný obraz používán žádným procesem. Když Linux pracuje s pamětí, mohou mu začít chybět fyzické stránky. V takovém případě provede snížení velikosti vyrovnávací paměti stránek.

3.8 Odkládání a rušení stránek

Když začne chybět fyzická paměť, musí se subsystém správy paměti pokusit o uvolnění fyzických stránek. Tento úkol plní odkládací démon jádra, *kswapd*.

Odkládací démon je speciální typ procesu, jde o vlákno jádra. Vlákna jádra jsou procesy, které nemají žádnou virtuální paměť a běží v režimu jádra ve fyzickém adresovém prostoru. Označení „odkládací démon" je poněkud zavádějící, protože tento démon toho dělá daleko více, než jen prosté odložení stránek do systémových odkládacích souborů. Jeho úkolem je zajistit v systému dostatek volných stránek, aby mohl systém správy paměti pracovat efektivně.

Odkládací démon jádra (*kswapd*) je spuštěn procesem *init* jádra při startu systému a sedí a čeká na periodické spouštění odkládacím časovačem jádra.

Vždy, když přijde zpráva od časovače, démon se podívá, zda v systému nezačínají chybět fyzické stránky. K rozhodování, zda uvolnit nějaké stránky, používá dvě proměnné, `free_pages_high` a `free_pages_low`. Dokud je počet volných stránek větší než hodnota `free_pages_high`, nedělá démon nic, spí dál, dokud jej časovač příště neprobudí. Pro potřeby těchto testů pracuje odkládací démon s počtem stránek, které jsou momentálně zapisovány do odkládacího souboru. Tento počet je uchováván v proměnné `nr_async_pages`, jejíž hodnota se inkrementuje vždy při zařazení stránky do fronty na zápis do odkládacího souboru a dekrementuje se vždy, když se dokončí zápis stránky na odkládací zařízení. Hodnoty `free_pages_low` a `free_pages_high` se nastavují při startu systému a odvozují se od celkového počtu fyzických stránek v systému. Pokud počet volných stránek klesne pod hodnotu `free_pages_high` nebo dokonce pod hodnotu `free_pages_low`, pokusí se odkládací démon třemi způsoby snížit počet obsazených fyzických stránek:

- Redukcí velikosti bufferů a vyrovnávací paměti stránek.
- Odložením sdílených paměťových stránek Systemu V.
- Odložením a zrušením stránek.

Pokud počet volných stránek klesne pod hodnotu `free_pages_low`, pokusí se odkládací démon uvolnit šest paměťových stránek. V opačném případě se pokouší uvolnit pouze tři stránky. Všechny tři výše uvedené metody se cyklicky opakují dokud není uvolněno dostatečné množství stránek. Odkládací démon si pamatuje, kterou metodu použil při posledním pokusu o uvolnění fyzických stránek. Při každém spuštění začíná uvolňovat stránky tou metodou, která posledně uspěla.

Jakmile uvolní dostatek stránek, odkládací démon usíná a čeká na další probuzení časovačem. Pokud byl důvodem pro uvolňování stránek pokles počtu volných stránek pod hodnotu `free_pages_low`, bude démon spát pouze polovinu svého normálního spacího času. Jakmile se počet volných stránek vyhoupne nad hodnotu `free_pages_low`, spí už démon mezi jednotlivými kontrolami svou normální dobu.

3.8.1 Redukce velikosti bufferů a vyrovnávací paměti stránek

Stránky držené ve vyrovnávacích pamětech bufferů a stránek jsou dobrými kandidáty na uvolnění ve vektoru `free_area`. Vyrovnávací paměť stránek, která obsahuje stránky paměťově mapovaných souborů, může obsahovat nepotřebné stránky, které zbytečně zaplňují paměť. Podobně buffery, obsahující data čtená a zapisovaná z/na fyzická zařízení, mohou obsahovat nepotřebné údaje. Když začnou docházet fyzické stránky, je poměrně jednoduché zrušit stránky ve vyrovnávacích pamětech, protože k tomu není nutné provádět zápis na fyzická zařízení

(což by bylo nutné při odkládání stránek z paměti). Zrušení takovýchto stránek nemá žádný výrazný vedlejší efekt kromě toho, že se zpomalí přístup k fyzickým zařízením a paměťově mapovaným souborům. Pokud se ovšem rušení stránek z vyrovnávacích pamětí provede spravedlivě, dotkne se to stejným dílem všech procesů.

Vždy, když se odkládací démon jádra pokouší snížit velikost vyrovnávacích pamětí, prozkoumá blok stránek ve stránkovém vektoru mem_map a zjistí, zda je možno nějaké stránky uvolnit z fyzické paměti. Velikost zkoumaných bloků stránek je tím vyšší, čím více démon odkládá, tedy čím níže poklesl počet volných fyzických stránek. Bloky stránek se zkoumají cyklicky, při každém pokusu o uvolnění stránek se prověřují jiné bloky. Tento postup se označuje jako *hodinový* algoritmus, protože podobně jako u pohybu minutové ručičky hodinek dojde vždy po několika voláních nakonec k prohlédnutí celého vektoru mem_map.

U každé zkoumané stránky se zjistí, zda je uložena ve vyrovnávací paměti stránek nebo v bufferu. Všimněte si, že v tomto okamžiku se neuvažuje o rušení sdílených stránek a že stránka nemůže být současně v obou vyrovnávacích pamětech. Pokud stránka není v žádné z vyrovnávacích pamětí, prozkoumá se následující stránka vektoru mem_map.

Stránky se ukládají ve vyrovnávacích bufferech (nebo přesněji, stránky v sobě obsahují buffery), čímž se zefektivňuje alokace a dealokace bufferů. Kód snížení počtu obsazených stránek se pokouší uvolnit buffery uložené ve zkoumané stránce.

Pokud se podaří uvolnit všechny buffery, je možno uvolnit i samotnou stránku. Pokud je taková stránka uložena ve vyrovnávací paměti stránek, odstraní se z ní a uvolní se.

Pokud se při tomto postupu podaří uvolnit dostatek fyzických stránek, čeká odkládací démon na další pravidelné probuzení. Protože žádná z uvolněných stránek nebyla součástí virtuálního adresového prostoru nějakého procesu (jednalo se o stránky vyrovnávacích pamětí), není nutné modifikovat tabulky stránek. Pokud se nepodaří uvolnit dostatek stránek vyrovnávacích pamětí, pokusí se odkládací démon odložit nějaké sdílené stránky.

3.8.2 Odkládání sdílených stránek Systemu V

Sdílená paměť Systemu V je meziprocesový komunikační mechanismus, kdy dva nebo více procesů sdílejí část své virtuální paměti, aby si mohly vzájemně předávat informace. Způsob, jakým se paměť mezi procesy sdílí, je podrobněji popsán v kapitole o meziprocesové komunikaci. Nám v této chvíli stačí vědět, že sdílená paměť Systemu V je popsána datovou strukturou shmid_ds. Tato struktura obsahuje ukazatel na seznam datových struktur vm_area_struct, jednu pro každý proces sdílející danou oblast virtuální paměti. Datové struktury vm_area_struct říkají, kam ve virtuálním adresovém prostoru daného procesu se má mapovat daná sdílená oblast paměti. Jednotlivé datové struktury vm_area_struct jsou

navzájem provázány ukazateli `vm_next_shared` a `vm_prev_shared`. Každá datová struktura `shmid_ds` navíc obsahuje seznam položek tabulek stránek, které popisují fyzické stránky, na něž se virtuální sdílené stránky mapují.

15 Odkládací démon jádra používá při odkládání sdílených stránek Systemu V hodinový algoritmus. Vždy, když je spuštěn, pamatuje si, kterou oblast sdílené virtuální paměti naposledy odložil. K tomu používá dva údaje, první je index do množiny datových struktur `schmid_ds`, druhý je index do seznamu položek tabulek stránek dané oblasti sdílené paměti. Tím je zajištěno, že stránky sdílené paměti jsou „obětovány“ spravedlivě.

Protože číslo rámce fyzické stránky sdílené virtuální stránky je uloženo v tabulkách stránek všech procesů, které danou oblast virtuální paměti sdílejí, musí odkládací démon jádra modifikovat všechny tyto tabulky aby se indikovalo, že stránka už není v paměti a je držena v odkládacím souboru. Při odkládání každé sdílené stránky musí odkládací démon nalézt záznam o této stránce v každé tabulce stránek všech procesů, které stránku sdílejí (záznam se nalezne podle ukazatele v datové struktuře `vm_area_struct`). Pokud je záznam o této sdílené stránce v tabulce stránek platný, démon jej označí jako neplatný a odložený a sníží počitadlo použití této sdílené stránky o jedničku. Záznam v tabulce stránek o odložené sdílené stránce obsahuje index do množiny struktur `shmid_ds` a index do položek tabulky stránek pro tuto oblast sdílené paměti.

Pokud počitadlo použití stránky dosáhne po modifikaci tabulek všech sdílejících procesů nulu, je možno sdílenou stránku zapsat do odkládacího souboru. Položka tabulky stránek v seznamu, na nějž ukazuje datová struktura `shmid_ds`, se pro tuto oblast sdílené paměti nahradí položkou odložené stránky. Položka odložené stránky je neplatná položka tabulky stránek, obsahuje ale index do množiny otevřených odkládacích souborů a offset toho souboru, v němž je možno odloženou stránku nalézt. Tyto informace se použijí jakmile bude nutné vrátit stránku zpět do fyzické paměti.

3.8.3 Odkládání a rušení stránek

16 Odkládací démon zkontroluje všechny procesy v systému a zjistí, zda některý je vhodným kandidátem na odložení.

Vhodným kandidátem jsou procesy, které je možno odložit (u některých to nejde) a které vlastní jednu či více stránek, jež je možno z paměti odložit nebo zrušit. Stránky se z fyzické paměti odloží do systémového odkládacího souboru pouze v tom případě, že je není možno obnovit nějakou jinou metodou.

Většina stránek spustitelného obrazu pochází ze souboru tohoto obrazu a je možno je jednoduše obnovit novým přečtením tohoto souboru. Například spustitelné instrukce se nikdy nemodifikují, takže se ani nikdy nezapisují do odkládacího souboru. Takovéto stránky je možno přímo zrušit - až se na ně proces bude příště odkazovat, načtou se do paměti zpátky přímo ze spustitelného obrazu.

Jakmile je nalezen proces k odložení, odkládací démon prohlédne všechny oblasti jeho virtuální paměti a hledá oblasti, které nejsou sdíleny ani uzamčeny.

Linux neodkládá všechny odložitelné stránky vybraného procesu, zruší pouze několik málo stránek.

Pokud jsou stránky v paměti uzamčeny, není možno je odložit ani zrušit.

Odkládací algoritmus Linuxu používá stárnutí stránek. Každá stránka má počitadlo (uložené v datové struktuře `mem_map_t`), které dává odkládacímu démonu jádra jakousi představu o tom, zda stránka stojí či nestojí za odložení. Stránky stárnou, když nejsou používány, a mládnou při každém přístupu. Odkládací démon odkládá pouze staré stránky. Implicitní operace při alokování nové stránky je, že stránka získá výchozí věk 3. Při každém přístupu se její věk inkrementuje o 3 až do maxima 20. Vždy když je odkládací démon spuštěn, nechá stránky zestárnout tím, že jejich věk dekrementuje o jednu. Toto implicitní chování je možno měnit, a proto se všechny tyto (a další odkládací informace) udržují ve struktuře `swap_control`.

Pokud je stránka stará (věk = 0), odkládací démon ji dále zpracuje. Modifikované stránky je nutné odložit. K popisu těchto vlastností stránek používá Linux strojově závislé příznaky ve strukturách PTE (viz obrázek 3.2). Ne všechny modifikované stránky je však nezbytně nutné uložit do odkládacího souboru. Každá oblast virtuální paměti procesu může mít svou vlastní odkládací operaci (na kterou ukazuje ukazatel `vm_ops` struktury `vm_area_struct`) a pak se použije tato operace. Pokud operace není definována, odkládací démon implicitně alokuje stránku v odkládacím souboru a zapíše rušenou stránku na odkládací zařízení.

Položka tabulky stránek pro odloženou stránku se přepíše údajem, který říká, že položka není platná, a obsahuje informace o tom, kde se stránka v odkládacím souboru nachází. Tento údaj obsahuje jednak offset odkládacího souboru, jednak určení, který odkládací soubor byl použit. Ať už byla použita jakákoliv metoda odkládání, původní fyzická stránka je uvolněna a přidá se zpět do seznamu `free_area`. Čisté (přesněji řečeno nemodifikované) stránky je možno zrušit a uložit zpět do struktury `free_area` okamžitě.

Pokud bylo odloženo a zrušeno dostatečné množství stránek, odkládací démon usíná. Při příštím probuzení posoudí jiný proces. Tímto způsobem odkládací démon „uzobne“ pár stránek každému procesu, dokud nebude mít systém opět dostatek paměti. Je to podstatně spravedlivější řešení než úplné odložení celého procesu.

3.9 Odkládací vyrovnávací paměť

Při odkládání stránek do odkládacích souborů se Linux snaží vyhnout zápisu, pokud to není nezbytné. Může nastat situace, kdy je stránka jak v odkládacím souboru, tak ve fyzické paměti. Dochází k tomu v případě, že stránka odložená z paměti byla později do paměti vrácena, když ji její proces opět potřeboval. Pokud nedošlo k zápisu do této stránky v paměti, její kopie v odkládacím souboru je stále platná.

Ke sledování těchto stránek používá Linux vyrovnávací odkládací paměť. Tato paměť představuje seznam položek tabulek stránek, jednu pro každou fyzickou stránku v systému. V položkách odložených stránek je zapsáno, ve kterém souboru jsou uloženy a jejich pozice v tomto souboru. Pokud je údaj ve vyrovnávací paměti nenulový, znamená to, že stránka uložená v odkládacím souboru nebyla modifikována. Pokud dojde k modifikaci stránky (zápisem do této stránky), údaj o ní se z odkládací vyrovnávací paměti odstraní.

Když Linux potřebuje fyzickou stránku odložil do odkládacího souboru, podívá se nejprve do odkládací vyrovnávací paměti, a pokud v ní najde platný záznam o stránce, nemusí ji zapisovat do odkládacího souboru. Platný záznam totiž indikuje, že stránka nebyla od posledního načtení z odkládacího souboru modifikována.

Položky v odkládací vyrovnávací paměti představují položky tabulky stránek pro odložené stránky. Jsou označeny jako neplatné, obsahují však informace, podle kterých Linux nalezne správný odkládací soubor a správnou stránku v tomto souboru.

3.10 Opětovné vkládání stránek

Modifikované stránky uložené v odkládacím souboru mohou být později znovu zapotřebí, například v okamžiku, kdy aplikace potřebuje zapisovat do té oblasti virtuální paměti, jejíž obsah je uložen v odložené fyzické stránce. Pokus o přístup k virtuální stránce, která není přítomna ve fyzické paměti, vyvolá výpadek stránky. Výpadkem stránky signalizuje procesor operačnímu systému, že není schopen přeložit virtuální adresu na fyzickou. V našem případě je to způsobeno tím, že při odložení stránky z paměti byl její záznam v tabulce stránek označen jako neplatný. Procesor není schopen obsloužit překlad virtuální adresy na fyzickou, takže předává řízení zpět operačnímu systému a říká mu, jaká virtuální adresa vypadla a jaký byl důvod výpadku. Formát těchto informací a způsob, jakým procesor předává řízení operačnímu systému, je závislý na procesoru.

Procesorově závislý kód obsluhy výpadku stránky musí nalézt datovou strukturu `vm_area_struct` popisující oblast virtuální paměti, do níž spadá vypadnuvší adresa. Dosáhne toho prohledáváním všech datových struktur `vm_area_struct` daného procesu tak dlouho,

dokud nenajde tu, která obsahuje adresu, jež způsobila výpadek. Jedná se o časově velmi kritický kód, a proto jsou datové struktury `vm_area_struct` procesu organizovány tak, aby vyhledávání zabralo co nejméně času.

Jakmile se obslouží všechny procesorově závislé akce a nalezne se oblast, do níž spadá vypadlá adresa, je už obsluha výpadku dále obecná a na procesoru nezávislá. **20**

Obecný kód obsluhy výpadku se podívá na záznam v tabulce stránek pro vypadlou stránku. Pokud položka indikuje, že stránka je odložena, musí ji Linux opětovně vložit zpět do fyzické paměti. Formát položky tabulky stránek pro odložené stránky je procesorově závislý, všechny procesory však položku označují jako neplatnou a ukládají v ní nějakou informaci o tom, kde odloženou stránku nalézt. Tyto informace Linux potřebuje, aby mohl stránku vrátit zpět do fyzické paměti.

V tomto okamžiku Linux zná adresu, která způsobila výpadek, a má položku tabulky obsahující informaci o tom, kam byla stránka odložena. Datová struktura `vm_area_struct` může obsahovat ukazatel na rutinu, která opětovně vloží jakoukoliv stránku dané oblasti virtuální paměti zpět do fyzické paměti. Jedná se o operaci *swapin*. Pokud daná oblast paměti má definovánu operaci *swapin*, Linux ji použije. To je mimo jiné i způsob, jakým se obsluhuje odkládání stránek sdílených mechanismem System V, protože jejich odkládání vyžaduje speciální obsluhu danou tím, že formát odložených sdílených stránek je poněkud odlišný od normálních odložených stránek. Operace *swapin* nemusí být definována a v takovém případě Linux předpokládá, že se jedná o běžnou stránku, která nevyžaduje speciální obsluhu. **21** **22** **23**

Nealokuje volnou fyzickou stránku a načte stránku zpět z odkládacího souboru. Informace o tom, kde se stránka v odkládacím souboru nachází (a ve kterém odkládacím souboru), se zjistí z položky tabulky stránek pro tuto stránku.

Pokud přístup, který způsobil výpadek stránky, nebyl pokusem o zápis, zůstává stránka ve vyrovnávací odkládací paměti a její položka v tabulce stránek je označena jako nezapisovatelná. Pokud následně dojde k zápisu do stránky, generuje se nový výpadek stránky a v tom okamžiku systém označí stránku jako modifikovanou a odstraní ji z vyrovnávací odkládací paměti. Pokud do stránky nebylo zapsáno a je zapotřebí ji znovu načíst, Linux si ušetří zápis stránky do odkládacího souboru, protože soubor už obsahuje platnou kopii stránky.

Pokud byl výpadek způsobem zápisem, stránka se odstraní z vyrovnávací paměti rovnou a označí se jako stránka modifikovaná a s povoleným zápisem.

Odkazy na zdrojové texty jádra

1 – Viz fs/buffer.c

2 – Viz mm/filemap.c

3 – Viz swap.h, mm/swap_state.c, mm/swapfile.c

4 – Viz include/asm/pgtable.h

5 – Viz include/linux/mm.h

6 – Viz __get_free_pages() v souboru mm/page_alloc.c

7 – Viz free_pages () v souboru mm/page_alloc.c

8 – Viz handle_mm_fault() v souboru mm/filemap.c

9 – Viz do_no_page() v souboru mm/memory.c

10 – Viz filemap_nopage() v souboru mm/filemap.c

11 – Viz include/linux/pagemap.h

12 – Viz kswapd() v souboru mm/vmscan.c

13 – Viz shrink_mmap() v souboru mm/filemap.c

14 – Viz try_to_free_buffer() v souboru fs/buffer.c

15 – Viz shm_swap() v souboru ipc/shm.c

16 – Viz swap_out() v souboru mm/vmscan.c

17 – Aby to mohl udělat, sleduje ukazatel vm_next na struktury vm area struct seřazené v seznamu mm_struct daného procesu.

18 – Viz swap_out() v souboru mm/vmscan.c

19 – Viz do_page_fault() v souboru arch/i386/mm/fault.c

20 – Viz do_no_page() v souboru mm/memory.c

21 – Viz do_swap_page() v soubrou mm/memory.c

22 – Viz shm_swap_in() v souboru ipc/shm.c

23 – Viz swap_in() v souboru mm/page_alloc.c

Kapitola

4

Procesy

V této kapitole je popsáno co to jsou procesy a jak jádro Linuxu procesy vytváří, spravuje a ruší.

Procesy provádějí v operačním systému úlohy. Program je souhrn strojových instrukcí a dat uložených ve spustitelném obrazu na disku a je to v zásadě pasivní entita. Naproti tomu proces můžeme chápat jako program v akci.

Jedná se o dynamickou entitu, která se trvale mění tak jak procesor vykonává jednotlivé strojové instrukce. Kromě instrukcí a dat programu jsou součástí procesu také čítač instrukcí a všechny registry procesoru a dále zásobník, který obsahuje dočasně odložená data jako parametry rutin, návratové adresy a uložené proměnné. Momentálně spuštěný program nebo proces zahrnuje všechny aktivity procesoru. Linux je multiprocesový operační systém. Procesy jsou oddělené úlohy, každý má vlastní práva a vlastní zodpovědnost. Pokud havaruje jeden proces, nezpůsobí to havárii jiného procesu v systému. Každý jeden proces běží ve svém vlastním virtuálním adresovém prostoru a není schopen komunikovat s ostatními procesy jinak než pomocí bezpečných, jádrem řízených mechanismů.

V době svého života proces používá řadu systémových prostředků. Používá procesor k vykonávání svých instrukcí a fyzickou paměť systému k uložení sebe sama a svých dat. Otevírá a používá soubory v souborovém systému a může přímo či nepřímo používat fyzická zařízení systému. Linux musí sledovat jednak proces a jednak prostředky, které proces využívá, aby mohl zajistit spravedlivé rozdělení prostředků mezi všemi procesy v systému. Nebylo by vůči ostatním procesům spravedlivé, kdyby si jeden proces monopolizoval většinu fyzické paměti nebo procesorového času.

Nejcennějším prostředkem systému je procesor, protože je v něm obvykle pouze jeden. Linux je multiprocesový operační systém a jeho cílem je zajistit, aby v každém okamžiku běžel na každém procesoru nějaký proces, aby se procesorů využilo co nejvíce. Pokud je procesů více než procesorů (což obvykle bývá), musejí ostatní procesy s během počkat než bude procesor volný. Multiprocessing je jednoduchá myšlenka - proces běží dokud nemusí čekat, obvykle na nějaký systémový prostředek, pokud prostředek má, může běžet dále. V jednoprocesovém systému, jako je například DOS, by procesor po dobu čekání na prostředek zůstal nečinný a mrhal by časem. V multiprocesovém systému je v paměti současně přítomno více procesů. Kdykoliv musí proces čekat, operační systém mu odebere procesor a předá jej jinému procesu, který jej potřebuje více. Rozhodování o tom, který proces má příště běžet, provádí plánovač úloh. Linux používá řadu různých plánovacích strategií, aby se zajistilo spravedlivé rozdělení procesoru.

Linux podporuje různé formáty spustitelných souborů, jedním je ELF, dalším může být Java a všechny tyto formáty musejí být spravovány transparentně tak, jak procesy využívají sdílené knihovny systému.

4.1 Procesy na Linuxu

1

Aby mohl Linux jednotlivé procesy v systému spravovat, je každý proces reprezentován datovou strukturou `task_struct` (termíny „úloha“ (task) a „proces“ jsou v Linuxu rovnocenné). Vektor `task` je pole ukazatelů na všechny struktury `task_struct` v systému.

Znamená to, že maximální počet procesů v systému je omezen velikostí vektoru `task`, který má implicitně 512 položek. Při vzniku procesu se v paměti alokuje nová struktura `task_struct` a přidá se vektoru `task`. Aby se usnadnilo vyhledávání, na aktuální proces je možno odkazovat se ukazatelem `current`.

Kromě normálních procesů Linux dále podporuje takzvané procesy reálného času. Tyto procesy musejí velmi rychle reagovat na externí události (proto procesy „reálného času“) a plánovač s nimi zachází odlišně od normálních uživatelských procesů. Přestože struktura `task_struct` je poměrně rozsáhlá a složitá, jednotlivé položky je možno rozdělit do několika funkčních oblastí:

Status

Při běhu procesu se podle okolností mění jeho *status*. Procesy v Linuxu mohou mít následující status[1]:

1 Neuvádím status SWAPPING, protože ten se zřejmě nepoužívá.

Running (běžící) Proces buď právě běží (aktuální proces), nebo je připraven k běhu (čeká na přidělení nějakého procesoru).

Waiting (čeká) Proces čeká na událost nebo prostředek. Linux rozlišuje dva typy čekajících procesů - *přerušitelné* a *nepřerušitelné*. Přerušitelné procesy je možno přerušit zasláním signálu, zatímco nepřerušitelné procesy čekají přímo na nějakou hardwarovou událost a není možno je přerušit za žádných okolností.

Stopped (zastaven) Proces byl zastaven, obvykle zasláním nějakého signálu. V zastaveném stavu se může nacházet například laděný proces.

Zombie Ukončený proces, který má ve vektoru `task` stále z nějakého důvodu zachovánu strukturu `task_struct`. Je to přesně to, co napovídá název, tedy mrtvý proces.

Plánovací informace

Tyto informace potřebuje plánovač k rozhodování, který proces si nejvíce zaslouží spustit.

Identifikátory

Každý proces má svůj identifikátor procesu. Identifikátor není indexem do vektoru `task`, je to prostě číslo. Každý proces má dále identifikátory uživatele a skupiny, které slouží k řízení přístupových práv procesu k souborům a zařízením v systému.

Meziprocesová komunikace

Linux podporuje klasické unixovské komunikační mechanismy jako signály, roury a semafory a dále mechanismy Systemu V pro sdílení paměti, semafory a fronty zpráv. Mechanismy pro meziprocesovou komunikaci jsou popsány v kapitole Meziprocesová komunikace.

Odkazy

V Linuxu není žádný proces nezávislý na ostatních procesech. Všechny procesy vyjma iniciálního procesu mají svůj rodičovský proces. Nové procesy nevznikají, kopírují se, nebo přesněji *klonují*, z předchozích procesů. Každá struktura `task_struct` každého procesu obsahuje ukazatele na jeho rodičovský proces a na jeho sourozence (ostatní procesy se stejným rodičovským procesem) a dále ukazatele na jeho synovské procesy. Rodinné vztahy mezi procesy v systému můžete zjistit pomocí příkazu `pstree`:

```
init(1)-+-crond(98)
        |-emacs(387)
        |-gpm(146)
        |-inetd(110)
        |-kerneld(18)
        |-kflushd(2)
        |-klogd(87)
        |-kswapd(3)
        |-login(160)---bash(192)---emacs(225)
        |-lpd(121)
        |-mingetty(161)
        |-mingetty(162)
        |-mingetty(163)
        |-mingetty(164)
        |-login(403)---bash(404)---pstree(594)
        |-sendmail(134)
        |-syslogd(78)
        '-update(166)
```

Navíc jsou všechny procesy v systému svázány obousměrně propojeným seznamem, jehož kořenem je datová struktura `task_struct` procesu `init`. Tento seznam jádru umožňuje dohled nad všemi procesy v systému. Navíc je potřebný pro podporu příkazů jako jsou `ps` nebo `kill`.

Časy a časovače

Jádro udržuje údaj o čase spuštění procesu i o celkovém času procesoru, který proces doposud spotřeboval. S každým tikem hodin jádro inkrementuje časový údaj o tom, kolik času proces strávil v systému a v uživatelském režimu. Linux navíc podporuje *intervalové* časovače procesů; proces může pomocí systémových volání tyto časovače nastavit tak, aby po uplynutí určité doby poslaly procesu signál. Tyto časovače mohou být jednorázové nebo periodické.

Souborový systém

Procesy mohou podle potřeby otevírat a zavírat soubory a struktura `task_struct` procesu obsahuje ukazatele na deskriptory otevřených souborů a dále ukazatele na dva VFS inody. Inody jednoznačně popisují soubor nebo adresář v souborovém systému a představují také uniformní rozhraní k nižším vrstvám souborového systému. Podpora souborových systémů v Linuxu je vysvětlena v kapitole Souborový systém. První ukazatel je na inode kořene procesu (jeho domovského adresáře), druhý ukazuje na jeho aktuální či pracovní (*pwd*) adresář.

Označení *pwd* je odvozeno od příkazu `pwd` systému Unix, který vypisuje pracovní adresář procesu. Pro inody je vedeno počitadlo `count`, které říká, že se na ně jeden či více procesů odkazují.

Virtuální paměť

Většina procesů má nějakou virtuální paměť (nemají ji pouze démony a vlákna jádra) a jádro Linuxu musí sledovat, jak se virtuální paměť mapuje na fyzickou paměť systému.

Procesorově specifický kontext

Proces můžeme chápat jako souhrn celého aktuálního stavu systému. Vždy, když proces běží, používá registry procesoru, zásobník a podobně. To všechno je kontext procesu a když dojde k pozastavení procesu, musí se celý kontext procesu uložit do struktury `task_struct`. Když plánovač proces opět spustí, kontext procesu se odtud obnoví.

4.2 Identifikátory

Linux, stejně jako systém Unix, používá ke kontrole přístupových práv k souborům identifikátory uživatelů a skupin. Každý soubor a adresář v Linuxu má své vlastníky a svá oprávnění, která popisují práva uživatelů systému k tomuto souboru či adresáři. Základními oprávněními jsou práva *čtení*, *zápisu* a *spuštění*, která se přiřazují třem třídám uživatelů: vlastníkovi souboru, procesům patřícím do nějaké skupiny a všem procesům v systému. Každá třída uživatelů může mít jiná oprávnění, takže je třeba možné nastavit oprávnění tak, že vlastník bude moci soubor číst i zapisovat, skupina souboru jej bude moci jenom číst a všechny ostatní procesy v systému nebudou mít žádná přístupová práva.

Skupiny představují způsob přiřazení privilegií k souborům a adresářům skupinám uživatelů a ne jen jednomu uživateli nebo všem uživatelům v systému. Můžete například vytvořit skupinu pro všechny uživatele podílející se na nějakém projektu a nastavit ji tak, že bude moci číst i modifikovat zdrojové kódy projektu. Proces může patřit do více skupin (maximální počet je implicitně 32) a tyto skupiny se ukládají ve vektoru `groups` struktury `task_struct` každého procesu. Pokud má nějaká skupina přístupová práva k nějakému souboru a proces do této skupiny patří, má k danému souboru práva této skupiny.

Ve struktuře `task_struct` procesu se udržují čtyři dvojice uživatelských a skupinových identifikátorů procesu:

uid, gid — Identifikátor uživatele a skupiny toho uživatele, jehož jménem proces běží.

efektivní uid a gid Existují některé programy, které mění uid a gid spouštějícího procesu na své vlastní (uložené jako atributy v inodu spustitelného obrazu). Tyto programy se označují jako *setuid* programy a jsou velmi užitečné, protože představují způsob jak omezit přístup ke službám, zejména službám spouštěným jménem někoho jiného, například síťových démonů. Efektivní uid a gid jsou nastavena podle atributů setuid programu, hodnoty uid a gid zůstávají nezměněny. Při kontrole přístupových práv používá jádro hodnoty efektivních uid a gid.

uid a gid souborového systému
Jsou normálně stejné jako efektivní uid a gid a slouží při kontrole přístupových práv k souborovému systému. Jsou nutné pro souborové systémy připojené prostřednictvím NFS, kdy server NFS v uživatelském režimu potřebuje přístup k souborům tak, jako kdyby byl určitým procesem. V takovém případě se změní uid a gid pouze souborového systému, ne efektivní uid a gid. Tím se zabrání situaci, kdy by zlomyslný uživatel mohl serveru NFS poslat signál *kill*. Signály *kill* se doručují procesům s určitým efektivním uid a gid.

uložené uid a gid Tyto hodnoty jsou vyžadovány normou POSIX a používají se v programech, které mění uid a gid procesu prostřednictvím systémových volání. Slouží k uložení skutečného uid a gid v době, kdy jsou původní hodnoty uid a gid změněny.

4.3 Plánování

Všechny procesy běží částečně v uživatelském režimu a částečně v systémovém režimu. Nízkoúrovňová hardwarová podpora těchto režimů může být různá, obecně ale platí, že existují nějaké bezpečnostní mechanismy při přechodu z uživatelského režimu do systémového a zpět. V uživatelském režimu má proces výrazně menší privilegia než v režimu systémovém. Vždy při použití systémového volání se proces přepíná z uživatelského režimu do systémového režimu a pokračuje v práci. V té době pracuje jménem procesu jádro. V Linuxu procesy nemohou vynuceně přerušit aktuálně běžící proces, nemohou jeho běh pozastavit, aby mohly běžet samy. Každý proces se dobrovolně vzdává procesoru, na němž běží v době, kdy musí čekat na nějakou systémovou událost. Proces řekněme čeká například na načtení znaku ze souboru. Toto čekání se provádí využitím systémového volání, v systémovém režimu. Proces použije knihovních funkcí k otevření a čtení souboru a poté následně používá systémových volání ke čtení bajtů z otevřeného souboru. V takovém případě je čekající proces pozastaven a umožní se běh jinému, potřebnějšímu procesu.

Procesy používají systémových volání často, takže mohou být také často přinuceny čekat. Ale i v této situaci může nějaký proces bez čekání běžet příliš dlouho a spotřebovávat nepřiměřené množství procesorového času, proto Linux používá preemptivní plánování úloh. Při využití tohoto schématu je každému procesu umožněno běžet malý objem času, 200 ms, a jakmile tento čas vyprší, naplánuje se běh jiného procesu a původní proces musí čekat na další příležitost ke běhu. Tento krátký přidělovaný časový úsek se označuje jako *časové kvantum (time-slice)*.

Rozhodování o tom, který proces právě nechat běžet, je úkolem *plánovače*.

Spustitelný proces je takový, který čeká pouze na přidělení procesoru. Linux používá rozumně jednoduchý na prioritě založený plánovací algoritmus, který volí aktuální procesy. Když zvolí nový aktuální proces, uloží status momentálně aktuálního procesu, procesorově specifické registry a další kontextové informace do datové struktury `task_struct`. Poté obnoví status nově naplánovaného procesu (což je opět procesorově závislá operace) a předá mu řízení systému. Aby mohl plánovač spravedlivě rozdělovat procesorový čas mezi spustitelné procesy v systému, ukládá si do struktury `task_struct` každého procesu následující informace:

policy Plánovací strategie používaná pro tento proces. Linux má dva typy procesů, normální a procesy reálného času. Procesy reálného času mají vyšší prioritu než všechny ostatní procesy. Pokud je k běhu připraven proces reálného času, bude vždy spuštěn. Procesy reálného času mohou používat dva typy strategie, `policy`, a to buď *round robin*, nebo *first in first out*. V plánování typu *round robin* se používá cyklická obsluha procesů a jednotlivé procesy se spouštějí v řadě za sebou. Při strategii *first in first out* se spustitelné procesy spouštějí v pořadí uvedeném ve spouštěcí frontě, které se nikdy nemění.

priority Priorita, kterou plánovač dává procesu. Je to zároveň množství času (v jednotkách zvaných `jiffies`[*]), které proces poběží, když bude naplánován. Prioritu procesů je možno měnit pomocí systémových volání a příkazem *renice*.

rt_priority Linux podporuje procesy reálného času, které se plánují tak, aby měly větší prioritu než všechny ostatní procesy v systému. Tato položka umožňuje plánovači přiřadit každému procesu reálného času jeho relativní prioritu. Prioritu procesů reálného času je možno měnit pomocí systémových volání.

counter Počet časových okamžiků (v `jiffies`), které proces může běžet. Při naplánování procesu se nastaví na hodnotu `priority` a s každým hodinovým tikem se dekrementuje.

[*] Poznámka překladatele: „vteřinka“, „momentík“.

Plánovač je spouštěn z několika míst v jádře. Spouští se po převedení aktuálního procesu do fronty čekajících procesů, může být zavolán po ukončení systémového volání, těsně před přepnutím procesu ze systémového režimu do uživatelského režimu. Dalším důvodem spuštění může být, že systémový časovač dodekrementoval hodnotu `counter` procesu na nulu. Při každém spuštění provede plánovač následující úkoly:

Úlohy jádra Plánovač spustí bottom-half obsluhu a zpracuje frontu úloh plánovače. Tato „odlehčená" vlákna jádra jsou podrobně popsána v kapitole Mechanismy jádra.

Aktuální proces Před výběrem dalšího procesu musí být spuštěn aktuální proces.

Pokud se používá plánovací strategie *round robin*, uloží se proces na konec fronty spouštěných procesů.

Pokud je úloha v přerušitelném (`INTERRUPTIBLE`) stavu a od posledního naplánování obdržela signál, přepne ji plánovač do stavu `RUNNING`.

Pokud vypršel aktuální proces, dostává se do stavu `RUNNING`.

Pokud je aktuální proces ve stavu `RUNNING`, zůstává v něm.

Procesy, které nejsou ani `RUNNING`, ani `INTERRUPTIBLE` se odstraní z fronty spouštěných procesů. Znamená to, že plánovač nebude tyto procesy posuzovat při výběru nejvhodnějšího procesu pro spuštění.

Výběr procesu Plánovač prohlédne procesy ve frontě spouštěných procesů a vybere nejlepšího kandidáta na spuštění. Pokud je ve frontě nějaký proces reálného času (proces s plánovací strategií reálného času), bude ohodnocen více než normální procesy. Váha normálního procesu odpovídá jeho hodnotě `counter`, u procesů reálného času to je `counter` plus 1000. Znamená to, že pokud je v systému nějaký spustitelný proces reálného času, bude spuštěn vždy dříve než normální spustitelné procesy. Aktuální proces, který vypotřeboval nějakou část svého časového kvanta (jeho hodnota `counter` byla snížena), je v nevýhodě před ostatními procesy se stejnou prioritou, což je v pořádku. Pokud má stejnou prioritu několik procesů, bude vybrán první proces z fronty. Aktuální proces bude zařazen na konec fronty. Ve vyváženém systému s mnoha procesy se stejnou prioritou se budou tyto procesy spouštět jeden po druhém. Takovéto plánování se označuje jako *round robin* – plánování cyklickou obsluhou. Jak ale procesy čekají na různé prostředky, pořadí jejich provádění se různě prohazuje.

Přepnutí procesů

Pokud je nejvhodnějším kandidátem na spuštění jiný proces než aktuální, musí být aktuální proces pozastaven a připraví se ke spuštění jiný proces. Když proces běží, používá registry a fyzickou paměť procesoru a systému. Při každém volání rutin jim v registrech předává parametry a může využívat hodnoty na zásobníku, například k uložení návratové adresy do volající rutiny. Když tedy plánovač běží, běží v kontextu aktuálního procesu. Jedná se sice o privilegovaný režim, režim jádra, ale stále běží jako aktuální proces. Když dojde k pozastavení tohoto procesu, je nutné uložit celý jeho strojový stav včetně čítače instrukcí (PC) a všech registrů procesoru do jeho struktury `task_struct`. Pak je nutné obnovit strojový stav nově naplánovaného procesu. Jedná se o strojově závislé operace, každý procesor to provádí trochu odlišným způsobem, obvykle však pro tuto operaci existuje nějaká hardwarová podpora.

Přepnutí procesů je poslední operace prováděná plánovačem. Uložený kontext předchozího procesu je tedy snímek hardwarového kontextu systému v okamžiku, kdy byl tento proces na konci plánovače. Obdobně tedy když dojde k nahrání nového procesu, v jeho snímku je rovněž zachycena situace, kdy se proces nacházel na konci plánovače, včetně obsahu instrukčního čítače a registrů.

Pokud předchozí proces nebo nový aktuální proces používají virtuální paměť, může být zapotřebí změnit údaje ve stránkovacích tabulkách. I tato akce je závislá na architektuře. Procesory jako Alpha AXP, které používají překladové tabulky a uložení položek tabulky stránek ve vyrovnávací paměti, musejí zneplatnit ty údaje ve vyrovnávací paměti, které patřily předchozímu procesu.

4.3.1 Plánování ve víceprocesorových systémech

Víceprocesorové systémy jsou ve světě Linuxu poměrně řídkavé, bylo však vynaloženo dost úsilí při budování Linuxu jako operačního systému typu SMP (Symmetric Multi-Processing). Symetrický multiprocesorový systém je takový, který je schopen stejnoměrně rozdělovat práci mezi všechny procesory v systému. Toto vyvažování je nejzřetelnější právě v plánovači.

Ve víceprocesorovém systému doufáme, že každý procesor provádí nějaký proces. Každý z procesorů spouští plánovač separátně podle toho, kdy jeho proces vyčerpá své časové kvantum nebo když musí čekat na systémové prostředky. První zajímavá věc v SMP systému je ta, že v něm není pouze jediný nečinný proces. V jednoprocesorovém systému je nečinným procesem první úloha ve vektoru `task`, v SMP systému existuje pro každý procesor jeden

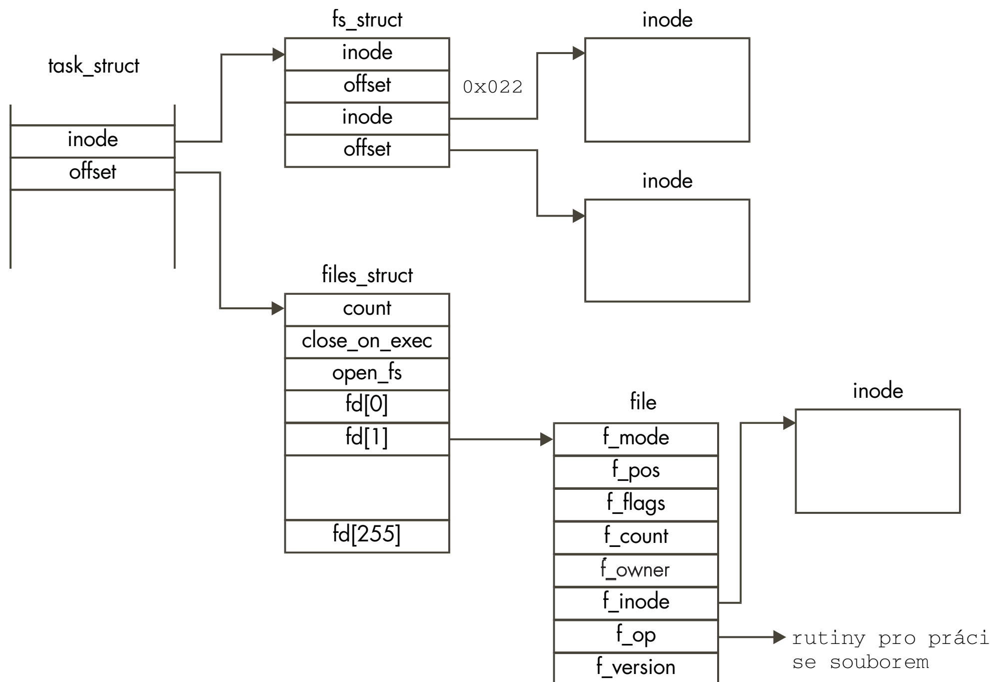
nečinný proces, navíc můžete mít několik nečinných procesorů. Navíc existuje jeden aktuální proces pro každý procesor, takže SMP systém musí vést záznamy o aktuálních a nečinných procesech pro každý procesor.

V SMP systému obsahuje struktura `task_struct` každého procesu číslo procesoru, na němž proces právě běží (`processor`), a rovněž číslo procesoru, na němž běžel naposledy (`last_processor`). Neexistuje žádný důvod, proč by proces nemohl při každém naplánování běžet na jiném procesoru, Linux však může provádění procesu omezit pouze na jeden nebo několik procesorů v systému pomocí masky `processor_mask`. Pokud je v ní nastaven N-tý bit, může proces běžet na N-tém procesoru. Když plánovač rozhoduje o naplánování nového procesu pro určitý procesor, nebude brát v úvahu ty, které nemají pro tento procesor nastaven příslušný bit v masce `processor_mask`. Plánovač navíc vždy částečně upřednostňuje procesy, které naposledy běžely na stejném procesoru, protože převedení procesu na jiný procesor s sebou nese jistou režii navíc.

4.4 Soubory

Na obrázku 4.1 vidíme, že v systému jsou pro každý proces dvě datové struktury, které popisují procesově specifické informace souborového systému. První z nich, struktura `fs_struct`, obsahuje ukazatele na VFS inody procesu a jeho hodnotu `umask`. Hodnota `umask` je implicitní hodnota pro vytváření nových souborů procesem a je možno ji změnit pomocí systémových volání.

Druhá datová struktura, struktura `files_struct`, obsahuje informace o všech souborech, které proces momentálně používá. Program čte ze *standardního vstupu* a zapisuje na *standardní výstup*. Všechna chybová hlášení se posílají na *standardní chybový výstup*. Fyzicky to mohou být soubory, terminálové linky nebo reálná zařízení, z pohledu programu jsou to ale vždy soubory. Každý soubor má svůj vlastní deskriptor a struktura `files_struct` obsahuje ukazatele na maximálně 256 datových struktur `file`, z nichž každá popisuje jeden soubor používaný procesem. Položka `f_mode` obsahuje režim vytvoření souboru: pro čtení, pro zápis i čtení nebo pouze pro zápis. `f_pos` obsahuje pozici v souboru, kde se provede následující operace čtení nebo zápisu. `f_inode` ukazuje na inode souboru a `f_ops` je ukazatel na vektor adres rutin, jedna pro každou funkci, kterou je možno se souborem provádět. Může to být například funkce pro zápis dat. Tato abstraktnost rozhraní je velmi mocná a umožňuje Linuxu podporovat široké spektrum různých typů souborů. Například roury se v Linuxu podporují, jak uvidíme později, právě tímto mechanismem.



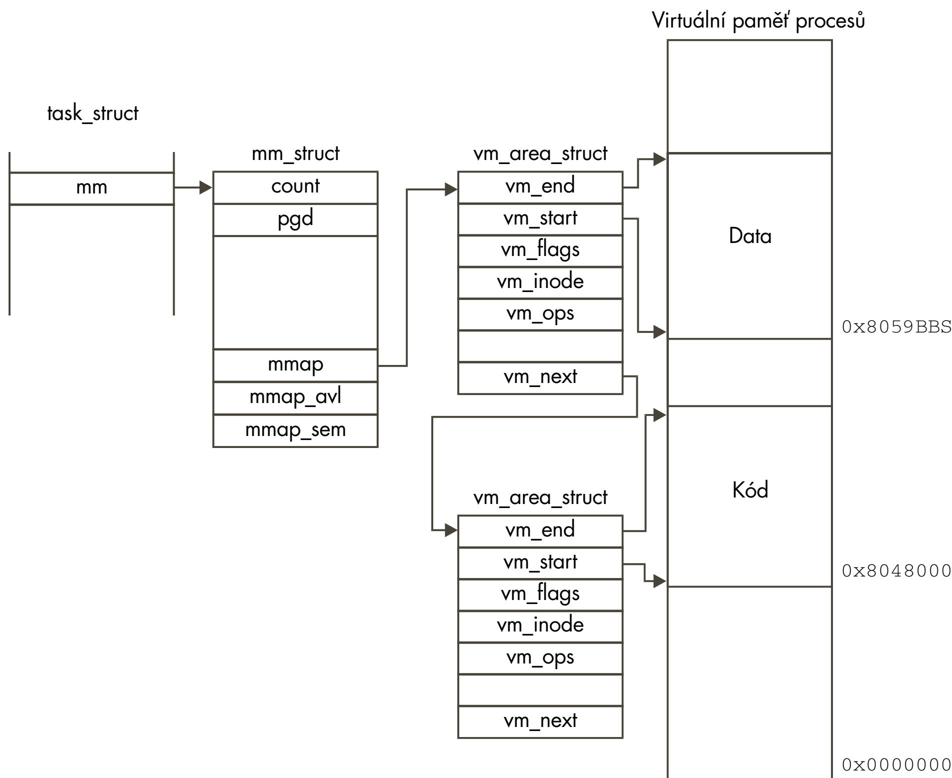
Obrázek 4.1
Soubory procesu

Vždy při otevření souboru se jeden z volných ukazatelů struktury `files_struct` použije jako ukazatel na novou strukturu `file`. Procesy v Linuxu očekávají při svém spuštění tři otevřené deskriptory souborů. Označují se jako *standardní vstup*, *standardní výstup* a *chybový výstup* a obvykle se dědí z vytvářejícího rodičovského procesu. Všechny přístupy k souborům se provádějí pomocí standardních systémových volání, která přebírají nebo vracejí deskriptory souborů. Tyto deskriptory jsou indexy do vektoru `fd` procesu, takže standardní vstup a výstup a chybový výstup mají deskriptory 0, 1 a 2. Všechny přístupy k souborům používají ke splnění svých požadavků operační rutiny datové struktury `file` spolu s inody VFS.

4.5 Virtuální paměť

Virtuální paměť procesu obsahuje spustitelný kód a data z mnoha zdrojů. Prvním zdrojem je nahraný obraz spustitelného programu, řekněme příkaz jako `ls`. Tento příkaz se, stejně jako všechny spustitelné obrazy, skládá jak ze spustitelného kódu, tak i z dat. Obraz obsahuje všechny informace potřebné k zavedení spustitelného kódu a s ním souvisejících dat do virtuální paměti procesu. Proces dále může alokovat (virtuální) paměť pro potřeby své práce, řek-

něme k uložení obsahu souborů, které načítá. Tato nově alokovaná virtuální paměť se musí navázat do stávající virtuální paměti procesu tak, aby ji bylo možno využít. Dále procesy používají knihovny obecně užitečných rutin, například rutin pro manipulaci se soubory. Nemá smysl, aby měl každý proces svou vlastní kopii knihovny, a proto Linux používá sdílené knihovny, které může používat více běžících procesů najednou. Kód a data těchto sdílených knihoven musí být rovněž přítomny ve virtuální paměti procesu a také všech ostatních procesů, které knihovnu používají.



Obrázek 4.2
Virtuální paměť procesu

V určitém časovém úseku proces nemusí používat veškerý kód a data obsažená ve své virtuální paměti. Může obsahovat části kódu, které se využívají jenom v určitých situacích, například v době inicializace nebo při zpracování určité události. Může využívat pouze některé rutiny ze sdílených knihoven. Bylo by plýtváním nahrávat do fyzické paměti veškerý kód a data, aby tam potom ležely nevyužity. Vynásobme takovéto plýtvání počtem procesů v systému a měli bychom systém, který by pracoval velmi neefektivně. Proto Linux používá techniku zvanou *stránkování na žádost*, kdy se virtuální paměť procesu zavádí do fyzické paměti pou-

ze v okamžiku, kdy ji proces potřebuje. Namísto přímého nahrání veškerého kódu a dat do paměti tedy Linux pouze modifikuje tabulku stránek procesu tak, že jednotlivé oblasti virtuální paměti označí jako používané, ale v paměti neexistující. Když se pak proces pokusí o přístup k těmto částem kódu či dat, hardware systému detekuje výpadek stránky a předá řízení jádru Linuxu, které situaci napraví. Proto musí Linux pro každou oblast virtuální paměti v adresovém prostoru procesu vědět, odkud kód pochází a jak jej do paměti dostat, aby mohl takovéto výpadky stránek ošetřit.

Jádro Linuxu tedy potřebuje spravovat všechny tyto oblasti virtuální paměti. Obsah virtuální paměti každého procesu je popsán datovou strukturou `mm_struct`, na kterou se ukazuje ze struktury `task_struct` procesu. Datová struktura `mm_struct` každého procesu obsahuje také informace o nahraném spustitelném obrazu a ukazatel na tabulku stránek procesu. Dále obsahuje ukazatele na seznam datových struktur `vm_area_struct`, z nichž každá reprezentuje jednu oblast virtuální paměti procesu.

Tento seznam je uspořádán vzestupně podle pořadí oblastí ve virtuální paměti. Na obrázku 4.2 vidíme rozvržení virtuální paměti jednoduchého procesu včetně datových struktur jádra, které virtuální paměť spravují. Protože jednotlivé oblasti virtuální paměti pocházejí z různých zdrojů, používá Linux abstraktní rozhraní, kdy struktura `vm_area_struct` obsahuje sadu ukazatelů (`vm_ops`) na rutiny obsluhy virtuální paměti. Díky tomu je možno celou virtuální paměť procesu obsluhovat konzistentním způsobem bez ohledu na to, že služby nižší úrovně se mohou pro jednotlivé oblasti lišit. Existuje zde například rutina, která se bude volat v případě, že se proces pokusí o přístup k oblasti paměti, která neexistuje; tímto mechanismem se obsluhují výpadky stránek.

Se seznamem struktur `vm_area_struct` jádro trvale pracuje, když vytváří nové oblasti virtuální paměti procesu nebo když opravuje odkazy na virtuální oblasti, které nejsou přítomny ve fyzické paměti procesu. Díky tomu se doba, strávená hledáním správné struktury `vm_area_struct`, stává kritickou pro celý výkon systému. Aby se přístup zrychlil, organizuje Linux struktury `vm_area_struct` také do takzvaného AVL (Adelson-Velskii a Landis) stromu. Strom je organizován tak, že každá struktura `vm_area_struct` (nebo též *uzel*) obsahuje levý a pravý ukazatel na své sousedy ve stromu. Levý ukazatel ukazuje na uzel s nižší počáteční virtuální adresou, pravý ukazatel ukazuje na uzel s vyšší počáteční virtuální adresou. Při hledání správného uzlu začne Linux od kořene stromu a pohybuje se z každého uzlu buď vlevo, nebo vpravo, až najde správný uzel. Pochopitelně nic není zadarmo a v tomto případě platíme za efektivní hledání vyšší náročností vložení nové struktury `vm_area_struct` do stromu.

Při alokování nové oblasti virtuální paměti Linux neprovádí skutečnou alokaci fyzické paměti. Pouze popíše nově vzniklou virtuální oblast vytvořením nové struktury `vm_area_struct`. Tato struktura se zapojí do seznamu oblastí virtuální paměti procesu. Když se proces pokusí o zá-

pis na virtuální adresu v nově vytvořené oblasti, dojde k výpadku stránky. Procesor se pokusí o dekódování virtuální adresy, protože však pro danou adresu neexistuje platná položka tabulky stránek, generuje výpadek stránky a přenechá řízení jádru systému. Linux se podívá, zda se požadovaná adresa nachází v existující oblasti virtuální paměti procesu. Pokud ano, vytvoří Linux patřičnou položku tabulky stránek a alokuje fyzickou paměťovou stránku. Pak může být nutné přenést stránku do fyzické paměti z diskového souboru nebo z odkládacího souboru. Poté je možno proces znovu spustit na stejné instrukci, která vyvolala výpadek stránky, a protože stránka už nyní existuje, proces může dále pokračovat.

4.6 Vytvoření procesu

Když se systém spustí, běží v režimu jádra a existuje pouze jediný, iniciální proces. Stejně jako všechny ostatní procesy, i iniciální proces má svůj strojový stav (kontext) reprezentovaný hodnotami registrů, zásobníkem a podobně. Když se vytvoří a spustí další procesy, bude tento stav uložen v datové struktuře `task_struct` iniciálního procesu. Na konci inicializace systému spustí iniciální proces vlákno jádra (zvané `init`) a pak nečinně čeká a nic nedělá. Kdykoliv není nic jiného na práci, spustí plánovač tento pozastavený proces. Struktura `task_struct` tohoto procesu jako jediná není alokována dynamicky, je staticky definována přímo v jádře a poněkud matoucně se jmenuje `init_task`.

Vlákno jádra či proces `init` má identifikátor procesu 1, protože se jedná o první faktický proces v systému. Provede nějaké počáteční nastavení systému (například otevření systémové konzoly a připojení kořenového souborového systému) a pak spustí inicializační program systému. Podle konkrétního systému se jedná o program `/etc/init`, `/bin/init` nebo `/sbin/init`. Při vytváření nových procesů v systému používá program `init` jako skriptový soubor `/etc/inittab`. Tyto nové procesy pak mohou samy vytvářet další nové procesy. Například proces `getty` může při pokusu o přihlášení vytvořit proces `login`. Všechny procesy v systému jsou (přímo či nepřímo) potomky procesu `init`.

Nové procesy se vytvářejí klonováním starých procesů, přesněji řečeno klonováním aktuálního procesu. Nová úloha se vytvoří systémovým voláním (*fork* nebo *clone*) a samotné klonování se odehraje v jádře v režimu jádra. Ve fyzické paměti systému se alokuje nová datová struktura `task_struct` a jedna nebo více fyzických stránek pro zásobník klonovaného procesu. Může se vytvořit identifikátor nového procesu, který musí být odlišný od identifikátorů všech existujících procesů. Je však možné, že nově vytvořený proces si ponechá identifikátor svého rodičovského procesu. Do vektoru `task` se vloží nová struktura `task_struct` a obsah struktury `task_struct` starého (aktuálního) procesu se zkopíruje do nové struktury.

Klonováním procesů umožňuje Linux dvěma procesům sdílet prostředky. Týká se to souborů, obsluhy signálů a virtuální paměti. Když se prostředky sdílejí, hodnota jejich počitadla `count` se zvyšuje, takže Linux příslušný prostředek neuvolní do té doby, dokud jej nepřestanou používat oba procesy. Pokud například klonovaný proces sdílí virtuální paměť se svým rodičem, jeho struktura `task_struct` bude obsahovat ukazatel na strukturu `mm_struct` rodičovského procesu a počitadlo `count` v této struktuře bude inkrementováno, aby se ukázalo, kolik procesů strukturu momentálně sdílí.

Klonování virtuální paměti je poněkud více rafinované. Musí se vytvořit nová sada datových struktur `vm_area_struct`, dále na ně ukazující struktura `mm_struct` a tabulka stránek nového procesu. V tomto okamžiku se ještě žádná virtuální paměť nekopíruje. Bylo by to dost obtížné a zdlouhavé vzhledem k tomu, že část virtuální paměti může ležet ve fyzické paměti, část ve spustitelných obrazech na disku a část třeba v odkládacím souboru. Namísto toho používá Linux techniku zvanou „kopírování při zápisu“, což znamená, že ke kopírování virtuální paměti dojde pouze v případě, že jeden z procesů do ní bude zapisovat. Virtuální paměť, do níž se nezapisuje (i když by k tomu mohlo dojít), se může mezi oběma procesy sdílet bez jakéhokoliv nebezpečí. Aby technika „kopírování při zápisu“ fungovala, jsou v tabulce stránek sdílené zapisovatelné stránky označeny jako „pouze pro čtení“ a ve struktuře `vm_area_struct` je uvedeno, že se jedná o stránce chráněné kopírováním při zápisu. Pokud se jeden z procesů pokusí o zápis do této oblasti virtuální paměti, dojde k výpadku stránky. Teprve v této fázi Linux vytvoří kopii paměti a upraví tabulky stránek a datové struktury virtuální paměti obou procesů.

4.7 Časy a časovače

Jádro si pamatuje čas vytvoření procesu a také objem času procesu, který proces po dobu svého života spotřeboval. S každým tikem hodin aktualizuje jádro čas, který proces existuje, a čas, který strávil v uživatelském režimu. Tyto časy se měří v jednotkách zvaných `jiffies`.

Kromě těchto účetních časovačů podporuje Linux ještě procesově závislé *intervalové* časovače.

Proces si může pomocí těchto časovačů nechat posílat různé signály. Linux podporuje tři typy časovačů:

reálné Tyto časovače běží v reálném čase a když doběhnou, proces dostane signál `SIGALARM`.

virtuální Tyto časovače běží pouze v době, kdy běží proces, a když doběhnou, dostane proces signál `SIGVTALARM`.

profilové Tyto časovače běží, pokud běží samotný proces nebo pokud systém jménem procesu vykonává nějakou činnost. Když časovače doběhnou, proces obdrží signál SIGPROF.

V dané chvíli může běžet jeden nebo více časovačů a Linux má všechny potřebné informace o časovačích uloženy v datové struktuře task_struct procesu. Pomocí různých systémových volání je možno časovače nastavovat, spouštět, zastavovat a přečíst jejich momentální hodnoty. Virtuální a profilové časovače se obsluhují stejným způsobem.

7

S každým tikem hodin se dekrementují časovače procesu a když doběhnou, pošle se příslušný signál.

8

Reálné časovače fungují poněkud jinak a při jejich obsluze používá Linux časovací mechanismy popsané v kapitole Mechanismy jádra. Každý proces má svou vlastní datovou strukturu timer_list a pokud používá reálné časovače, přidávají se do fronty systémových časovačů. Když časovač doběhne, odstraní jej bottom-half handler z fronty a zavolá obsluhu intervalového časovače.

9

Tím se generuje signál SIGALARM, časovač se restartuje a znovu se přidává do fronty systémových časovačů.

4.8 Spouštění programů

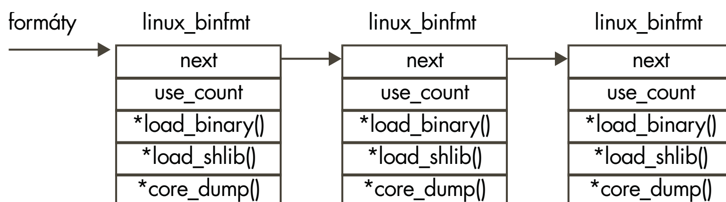
V Linuxu se, stejně jako v Unixu, programy a příkazy normálně spouštějí pomocí interpretu příkazů. Příkazový interpret je uživatelský proces jako každý jiný a označuje se termínem *shell* [3].

V Linuxu existuje mnoho příkazových interpretů, nejpopulárnějšími jsou sh, bash a tcsh. S výjimkou několika vestavěných příkazů jako cd nebo pwd jsou ostatní příkazy spustitelné binární soubory. Při zadání příkazu prohledává příkazový interpret adresáře ve *vyhledávací cestě* procesu, uložené v proměnné prostředí PATH, a hledá spustitelný obraz zadaného jména. Pokud soubor najde, nahraje jej a spustí. Příkazový interpret vytvoří svůj klon pomocí výše popsaného mechanismu *fork* a nový proces nahradí binární obraz příkazového interpretu obsahem právě nahraného spustitelného obrazu. Za normálních okolností příkazový interpret čeká na dokončení příkazu, přesněji řečeno na skončení synovského procesu. Příkazový interpret můžete znovu vyvolat tím, že synovský proces přesunete do pozadí stiskem CONTROL-Z, čímž se synovskému procesu pošle signál SIGSTOP a proces se zastaví. Příkazem bg pří-

3 Představme si jádro jako základ a kolem něj je obalena ulita (shell), která představuje uživatelské rozhraní.

kazového interpretu pak můžete přesunout do pozadí příkazový interpret, který synovskému procesu pošle signál `SIGCONT` a tak obnoví jeho běh. Příkazový interpret pak zůstává v pozadí, dokud synovský proces neskončí nebo dokud nepotřebuje terminálový vstup či výstup.

Spustitelný soubor může mít mnoho formátů nebo se může jednat o skript. Skriptové soubory je nutné rozeznat a nechat je provést příslušným interpretem, například skripty příkazového interpretu se provádějí programem `/bin/sh`. Spustitelné objektové soubory obsahují spustitelný kód a data doplněná o další informace, které umožňují operačnímu systému nahrát soubor do paměti a spustit jej. Nejběžnějším formátem spustitelného souboru v Linuxu je formát ELF, teoreticky je však Linux natolik pružný, že může obsloužit prakticky jakýkoliv objektový formát.



Obrázek 4.3
Registrované binární formáty

Stejně jako u souborových systémů jsou i binární formáty buď součástí jádra Linuxu nebo se dají dohrát jako moduly. Jádro udržuje seznam podporovaných binárních formátů (viz obrázek 4.3) a když dojde k pokusu o spuštění souboru, vyzkouší se každý binární formát dokud některý z nich nebude fungovat.

Linux běžně podporuje formáty `a.out` a `ELF`. Spustitelné soubory nemusejí být nahrány v paměti celé, používá se metoda zvaná vynucené nahrávání. Jednotlivé části spustitelného obrazu se zavádějí do paměti podle toho, jak je proces potřebuje. Nepoužívané části je možno z paměti uvolnit.

4.8.1 ELF

Objektový souborový formát ELF (Executable and Linkable Format) navržený v Unix System Laboratories je dnes zaveden jako nejčastěji používaný formát v Linuxu. Přestože v porovnání s jinými objektovými formáty jako `ESCOFF` a `a.out` má tento formát poněkud vyšší výkonnostní režii, je daleko pružnější. Spustitelné soubory ve formátu ELF obsahují spustitelný kód, někdy označovaný jako *text*, a *data*. Tabulky ve spustitelném obraze říkají, jak má být proces umístěn do virtuální paměti procesu. Staticky linkované obrazy jsou sestaveny linkerem (`ld`) do jediného obrazu, který obsahuje veškerý kód a data potřebná ke spuštění tohoto obrazu. Obraz dále specifikuje své rozvržení v paměti a adresu, od níž se má kód začít vykonávat.

Spustitelný obraz ELF

	e_ident	'E' 'L' 'F'
	e_entry	0x8048090
	e_phoff	52
	e_phentsize	32
	e_phnum	2
Fyzická hlavička	p_type	PT_LOAD
	p_offset	0
	p_vaddr	0x8048000
	p_filesz	68532
	p_memsz	68532
	p_flags	PF_R, PF_X
Fyzická hlavička	p_type	PT_LOAD
	p_offset	68536
	p_vaddr	0x805BB8
	p_filesz	2200
	p_memsz	4248
	p_flags	PF_R, PF_W
	Kód	
	Data	

Obrázek 4.4
Souborový formát ELF.

Na obrázku 4.4 vidíme uspořádání staticky linkovaného spustitelného obrazu ve formátu ELF.

Jedná se o jednoduchý program v jazyce C, který vytiskne text „hello world“ a ukončí se. Hlavička souboru říká, že se jedná o obraz ELF se dvěma fyzickými hlavičkami (hodnota `e_phnum` je 2), které začínají 52 bajtů (`e_phoff`) od počátku souboru. První fyzická hlavička popisuje spustitelný kód obrazu. Patří na virtuální adresu *0x8048000* a je celkem 65 532 bajtů dlouhý. Velikost je dána tím, že jde o staticky linkovaný obraz, který obsahuje celý knihovní kód funkce `printf()`. Vstupní bod obrazu, tedy adresa první instrukce programu, není na počátku obrazu, ale na virtuální adrese *0x8048090* (položka `e_entry`). Kód začíná ihned za druhou fyzickou hlavičkou. Tato druhá hlavička popisuje data programu a nahrává se do virtuální paměti na adresu *0x8059BB8*. Data je možno číst i zapisovat. Můžete si všimnout, že velikost dat je 2 200 bajtů (`p_filesz`), zatímco velikost paměti dat je 4 248 bajtů. Je to dáno tím, že prvních 2 200 bajtů obsahuje předinicializovaná data, zatímco zbývajících 2 048 bajtů obsahuje data, která se budou inicializovat až při běhu kódu.

Když Linux nahrává spustitelný obraz ve formátu ELF do virtuálního paměťového prostoru procesu, neprovádí skutečné nahrávání obrazu.

Nastaví datové struktury virtuální paměti, strom struktur `vm_area_struct` a tabulky stránek. Když se program začne provádět, dojde k výpadku stránky, což způsobí, že se kód a data programu načtou do fyzické paměti. Nepoužité části programu se do paměti nikdy nenahrají. Jakmile zavaděč binárního formátu ELF zjistí, že nahrávaný obraz je platným obrazem ve formátu ELF, odstraní z virtuální paměti procesu obraz stávajícího prováděného programu. Protože proces je klonem obrazu (všechny procesy jsou klonem), starý obraz je obraz programu prováděného rodičovským procesem, například tedy obrazem interpretu příkazů. Odstranění starého spustitelného obrazu zruší staré struktury virtuální paměti a tabulky stránek. Dojde rovněž k vymazání všech handlerů signálů a k zavření všech otevřených souborů. Na konci rušení je proces připraven přijmout nový spustitelný obraz. Bez ohledu na formát spustitelného obrazu se struktura `mm_struct` procesu nastavuje vždy stejnými informacemi. Musí totiž obsahovat ukazatele na začátek a konec kódu a dat obrazu. Tyto hodnoty se načtou z fyzických hlaviček formátu ELF a jimi určené oblasti programu se mapují do virtuálního adresového prostoru procesu. Zde se rovněž nastaví datové struktury `vm_area_struct` a provede se inicializace tabulky stránek procesu. Datová struktura `mm_struct` dále obsahuje ukazatele na parametry předávané programu a na proměnné prostředí procesu.

Sdílené knihovny ve formátu ELF

Dynamicky linkované obrazy neobsahují veškerý kód a data, která ke své činnosti potřebují. Některé části jsou umístěny ve sdílených knihovnách, které se zavádějí až v okamžiku spuštění programu. Tabulky sdílených knihoven formátu ELF dále slouží *dynamickému linkeru* při linkování sdílených knihoven do běžícího programu. Linux používá několik dynamických linkerů, `ld.so.1`, `libc.so.1` a `ld-linux.so.1`, všechny jsou umístěny v adresáři `/lib`. Knihovny obsahují běžně používaný kód, například internacionální podporu. Bez použití dynamického linkování by každý program musel obsahovat vlastní kopii těchto knihoven a bylo by tak zapotřebí podstatně více diskového prostoru a virtuální paměti. Při dynamickém linkování jsou v tabulkách obrazu ELF uloženy informace pro každou knihovní funkci, na níž se program odkazuje. Tyto informace dynamickému linkeru říkají jak nalézt knihovní funkci a jak ji vlinkovat do adresového prostoru programu.

4.8.2 Skriptové soubory

Skriptové soubory jsou spustitelné soubory, které ke svému běhu potřebují interpret. V Linuxu existuje celá řada interpretů, například `wish`, `perl` a příkazové interprety jako `tcsh`. Linux používá standardní konvenci systému Unix, kdy první řádek skriptu obsahuje jméno interpretu. Typický skript bude tedy začínat třeba takto:

```
#!/usr/bin/wish
```

13 Zavaděč skriptu se pokusí nalézt interpret skriptu.

Provádí to tak, že se pokusí otevřít spustitelný soubor, uvedený v prvním řádku skriptu. Pokud se mu jej podaří otevřít, má ukazatel na jeho VFS inode a může přejít k dalšímu kroku a nechat interpret vykonat skriptový soubor. Jméno skriptového souboru je parametrem číslo nula (tedy prvním parametrem) interpretu a všechny ostatní parametry se posouvají o jednu pozici vzad (původně první parametr se stává druhým a tak dále). Nahrání interpretu se provádí stejným mechanismem, jakým Linux nahrává všechny spustitelné soubory. Linux opět zkouší všechny registrované binární formáty dokud nenalezne vyhovující formát. Znamená to, že teoreticky můžete na sebe stavět několik interpretů a binárních formátů. Obsluha spustitelných souborů v Linuxu je tedy velice pružná.

Odkazy na zdrojové texty jádra

1 – `Viz include/-linux/sched.h`
2 – `Viz schedule()in kernel/sched.c`
3 – `Viz schedule()in kernel/sched.c`
4 – `Viz include/-linux/sched.h`
5 – `Viz do_fork() in kernel/fork.c`
6 – `Viz kernel /itimer.c`
7 – `Viz do_it_virtual() in kernel/sched.c`
8 – `Viz do_it_prof() in kernel /sched.c`
9 – `Viz it_real_fn() in kernel/itimer.c`
10 – `Viz do_execve() in fs/exec.c`
11 – `Viz include/-linux/elf.h`
12 – `Viz do_load_elf_binary() in fs/binfmt_elf.c`
13 – `Viz do_load_script() in fs/-binfmt_script.c`

Kapitola

5

Meziprocesorová komunikace

Při vzájemné koordinaci svých aktivit komunikují procesy navzájem mezi sebou a jádrem. Linux podporuje řadu mechanismů pro meziprocesovou komunikaci (IPC). Dvěma z nich jsou signály a roury, Linux však dále podporuje IPC mechanismy Systemu V, pojmenované podle verze Unixu, kde byly poprvé zavedeny.

5.1 Signály

Signály jsou jednou z nejstarších metod meziprocesové komunikace zavedených v systémech Unix. Slouží k signalizaci asynchronních událostí jednomu nebo více procesům. Signál může být generován přerušením klávesnice nebo chybovými okolnostmi, jako například pokusem o přístup k neexistující oblasti virtuální paměti. Signály jsou rovněž využívány v příkazových interpretech k signalizaci řídících příkazů v synovských procesech.

Je definována jakási množina signálů, které může generovat jádro nebo jiné procesy za předpokladu, že mají dostatečná privilegia. Seznam všech existujících signálů můžete získat pomocí příkazu `kill` (`kill -l`), na mém systému (s procesorem Intel) dostávám následující seznam:

```
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGIOT       7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR
```

Pro platformu Alpha se signály mohou lišit. Proces se může rozhodnout, že většinu generovaných signálů bude ignorovat. Existují ale dvě výjimky: signál `SIGSTOP`, který způsobení pozastavení činnosti procesu, a signál `SIGKILL`, který způsobí ukončení procesu. Ve všech ostatních případech může proces pro signály zvolit libovolnou obsluhu. Procesy mohou signály zablokovat nebo, pokud je neblokují, mohou zvolit vlastní obsluhu nebo mohou obsluhu signálu ponechat na jádru. Pokud je signál obsluhován jádrem, používá jádro implicitní obslužné mechanismy příslušného signálu. Například implicitní akce při obsluze signálu `SIGFPE` (výjimka operace v pohyblivé řádové čárce) je vytvoření souboru `core` a ukončení procesu. Signály nemají žádnou vzájemnou prioritu. Pokud dojde ve stejném okamžiku k vygenerování dvou signálů, mohou se v procesu objevit v jakémkoliv pořadí. Obdobně neexistuje žádný mechanismus pro obsluhu více stejných signálů. Neexistuje způsob jak může proces říci, zda obdržel jeden signál `SIGCONT` nebo 42 těchto signálů.

Linux implementuje signály pomocí informací uložených ve struktuře `task_struct` procesu. Počet podporovaných signálů je omezen velikostí slova procesoru. Procesory se slovem o velikosti 32 bitů mohou mít maximálně 32 různých signálů, zatímco 64bitové procesory, jako například Alpha AXP, mohou mít až 64 signálů. Momentálně aktivní signály jsou uloženy v položce `signal` s maskou blokovaných signálů uloženou v položce `blocked`. S výjimkou signálů `SIGSTOP` a `SIGKILL` je možno blokovat všechny signály. Pokud je generován blokovaný signál, zůstává platný až do doby, než dojde k jeho odblokování. Linux dále udržuje informace o obsluze všech možných signálů, které jsou uloženy v datové struktuře `sigaction`, na níž ukazuje struktura `task_struct`. Kromě dalšího obsahuje buď adresu obslužné rutiny signálu, nebo příznak, který Linuxu říká, zda si proces přeje signál ignorovat, nebo zda jej má obsluhovat jádro. Proces může implicitně nastavenou obsluhu signálů modifikovat pomocí různých systémových volání, která následně modifikují strukturu `sigaction` a také hodnotu masky `blocked`.

Ne každý proces v systému může posílat signály všem ostatním procesům, to může pouze jádro a superuživatel. Normální procesy mohou posílat signály pouze procesům se stejným *uid* a *gid* nebo procesům ve stejné skupině procesů. Signály jsou generovány nastavením příslušného bitu v položce `signal` struktury `task_struct`. Pokud proces signál nezablokoval a čeká v přerušitelném stavu, dojde k jeho probuzení přepnutím stavu na spuštěný a zajištěním, že se bude nacházet ve frontě spouštěných procesů. Díky tomu jej bude plánovač v dalším kole plánovaní považovat za kandidáta na spuštění. Pokud je požadována implicitní obsluha, je Linux schopen obsluhu signálu optimalizovat. Pokud se například objeví signál `SIGWINCH` (změna fokusu X okna) a požaduje se implicitní obsluha, pak není potřeba udělat vůbec nic.

Signály se procesům nepředávají bezprostředně po jejich výskytu, musejí počkat až do příštího spuštění procesu. Vždy když proces opouští systémové volání, kontrolují se jeho položky `signal` a `blocked` a pokud se v nich nachází nějaký neblokovaný signál, je možno jej ny-

ní doručit. Může se to jevit jako velmi nespolehlivá metoda, avšak každý proces v systému trvale používá systémová volání, například při zápisu znaku na terminál. Proces se může v případě potřeby rozhodnout čekat na signál, pak je uveden do pozastaveného, nicméně přerušitelného stavu až do doby, než se signál objeví. Kód obsluhy signálů v Linuxu pak pro každý neblokovaný přítomný signál vyhodnocuje údaje ve struktuře `sigaction`.

Pokud je obsluha signálu nastavena na implicitní obsluhu, obslouží signál jádro. Implicitní obsluha signálu `SIGSTOP` převede proces do zastaveného stavu a nechá plánovač spustit další proces. Implicitní obsluha signálu `SIGFPE` provede výpis jádra a ukončí proces. Proces však může zvolit vlastní obsluhu signálu. Jedná se o rutinu, jejíž adresa je uložena ve struktuře `sigaction` a která se bude volat v okamžiku výskytu signálu. Jádro musí zavolat obslužnou rutinu procesu, což je akce závislá na konkrétní platformě, systém se ale vždy musí vyrovnávat s faktem, že proces se momentálně nachází v režimu jádra a následujícím krokem je návrat do uživatelského režimu, z nějž byla volána nějaká systémová rutina nebo rutina jádra. Tento problém se řeší pomocí manipulace se zásobníkem a registry procesu. Ukazatel instrukcí programu se nastaví na adresu rutiny obsluhy signálu a parametry předávané rutině se uloží na zásobník nebo zapíší do příslušných registrů. Když proces obnoví svou činnost, vypadá to, jako kdyby byla obslužná rutina signálu volána běžnými mechanismy.

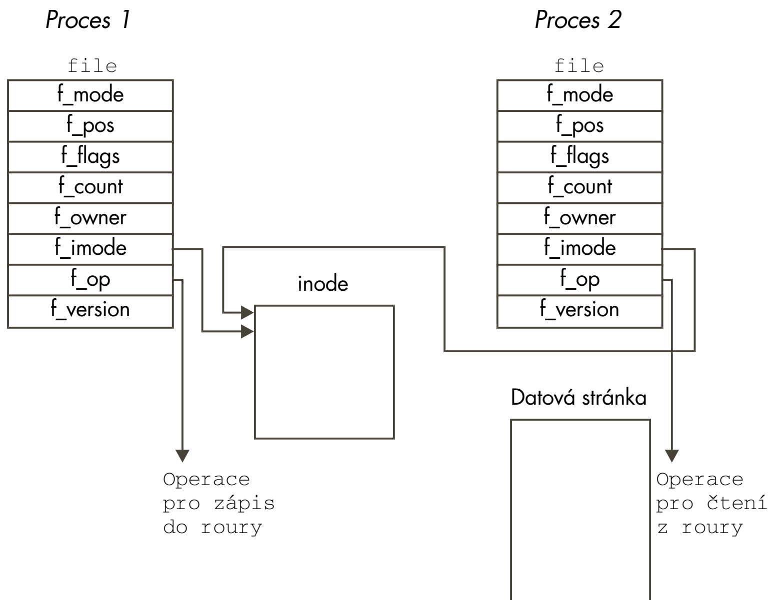
Linux je kompatibilní s normou POSIX, takže procesy mohou při volání obslužné rutiny signálu měnit blokování signálů. Obnáší to změnu masky `blocked` při volání obsluhy signálu. Maska `blocked` musí být po skončení obslužné rutiny nastavena zpět na původní hodnotu. Linux proto na zásobník přidává adresu úklidové rutiny, která zajistí obnovu původní hodnota masky. Linux rovněž provádí optimalizaci v případě, kdy se musí volat více obslužných rutin, a to tak, že jejich adresy najednou naskládá na zásobník, takže když jedna obsluha skončí, vrací se přímo do další obslužné rutiny a až poslední skáče do úklidové rutiny.

5.2 Roury

Všechny klasické příkazové interprety Linuxu podporují přesměrování. Například:

```
$ ls | pr | lpr
```

Tento příkaz předává rourou výpis obsahu adresáře pořízený příkazem **`ls`** na standardní vstup příkazu **`pr`**, který jej rozděluje na stránky. Nakonec se standardní výstup příkazu **`pr`** předává na standardní vstup příkazu **`lpr`**, který výsledek vytiskne na tiskárně. Roury jsou jednosměrné bajtové proudy, které propojují standardní výstup jednoho procesu se standardním vstupem druhého procesu. Žádný z procesů si tohoto přesměrování není vědom a všem funguje jako normálně. Dočasné roury mezi procesy obsluhuje příkazový interpret.



Obrázek 5.1
Roury

1 V Linuxu se roury implementují pomocí dvou datových struktur `file`, které obě ukazují na stejný dočasný inode VFS, který sám o sobě ukazuje na fyzickou stránku v paměti. Na obrázku 5.1 vidíme, že každá ze struktur `file` obsahuje ukazatele na rozdílné vektory souborových rutin; jedna obsahuje ukazatel na operaci zápisu do roury, druhá ukazatel na operaci čtení z roury.

Tím se před obecnými systémovými voláními zajišťujícími čtení a zápis do souborů skrývají implementační detaily. Když první proces zapisuje do roury, kopírují se bajty na sdílenou datovou stránku, když druhý proces čte z roury, bajty se kopírují ze sdílené stránky. Linux musí synchronizovat přístup k rouře. Musí zajistit, aby čtenář i písař roury byli v koordinaci, k zajištění jejich zamykání používá fronty a signály.

2 Když písař zapisuje do roury, používá standardní knihovní funkce zápisu. Všem těmto funkcím se předávají deskriptory, které jsou indexy do množiny datových struktur `file` procesu, každý deskriptor reprezentuje jeden otevřený soubor nebo, jako v tomto případě, otevřenou rouru. Zápisová rutina pak používá k zajištění požadavků na zápis informace uložené v inodu VFS, který reprezentuje danou rouru.

Dokud je dostatek místa k zápisu bajtů do roury a dokud není roura zamčena čtenářem, Linux ji zamyká pro písaře a kopíruje bajty zapisované z adresového prostoru písaře do sdílené datové stránky. Když si rouru zamkne čtenář nebo když v ní není dostatek místa pro data, aktuální proces se uspí a čeká v čekací frontě inodu roury. Plánovač pak naplánuje spuštění jiného procesu. Proces je přerušitelný, takže může přijímat signály a bude čtenářem probuzen až bude dostatek místa pro zápis dat nebo když dojde k odemčení roury. Po zapsání všech dat se inode roury odemkne a bude probuzen čtenář, čekající ve frontě tohoto inodu.

Čtení dat z roury je velmi podobné jejich zápisu. Procesům je povoleno neblokovací čtení (závisí to na režimu otevření souboru či roury) a v takovém případě pokud nejsou žádná data k načtení nebo pokud je roura uzamčená, vzniká chyba. Znamená to, že proces může pokračovat. Druhá možnost je čekat ve frontě inodu roury tak dlouho, dokud zápisová operace neskončí. Když oba procesy skončí s používáním roury, zruší se její inode i sdílená datová stránka.

Linux rovněž podporuje *pojmenované* roury, zvané také FIFO, protože zřetězují operace na principu First In, First Out. První data zapsaná do roury budou také jako první přečtena. Na rozdíl od normálních rour nejsou FIFO dočasné objekty, představují trvalé entity v souborovém systému a vytvářejí se příkazem `mkfifo`. Procesy mohou FIFO používat, pokud k nim mají příslušná přístupová práva. Způsob otevření FIFO se poněkud liší od rour. Roura (tedy její dvě datové struktury `file`, její inode a sdílená datová stránka) se vytvářejí jednorázově za běhu, zatímco FIFO existuje trvale a uživatelé si je otevírají a zavírají. Linux musí ošetřovat situace, kdy čtenář otevře FIFO dříve než písař nebo kdy se čtenář pokouší číst, přestože písař ještě nic nezapsal. Až na tyto výjimky se FIFO obsluhují prakticky úplně stejně jako roury a používají stejné datové struktury a operace.

5.3 Sokety

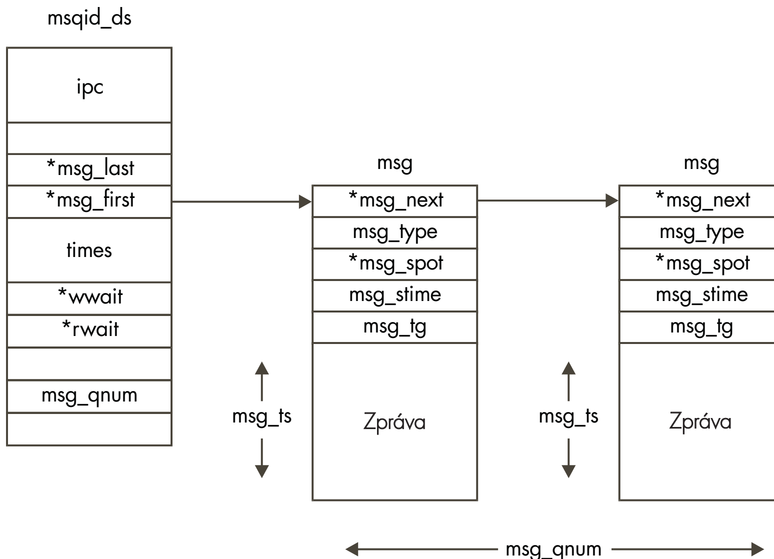
5.3.1 IPC mechanismy Systemu V

Linux podporuje tři mechanismy pro meziprocesovou komunikaci, které se poprvé objevily v Unixu System V v roce 1983. Jedná se o fronty zpráv, semafory a sdílenou paměť. Tyto IPC mechanismy Systemu V všechny používají shodné autentifikační metody. Procesy mohou k těmto prostředkům přistupovat pouze když jádru pomocí systémových volání předají jedinečný referenční identifikátor. Přístup k IPC objektům Systemu V se řídí pomocí přístupových oprávnění velmi podobně, jako se řídí přístup k souborům. Přístupová práva k IPC objektům Systemu V nastavuje tvůrce objektu pomocí systémových volání. Referenční identifikátor objektu používají všechny tyto mechanismy jako index do tabulky prostředků. Nejde ovšem přímo o index, k vygenerování indexu jsou nutné ještě pomocné výpočty.

4 Všechny datové struktury reprezentující IPC objekty Systemu V se v Linuxu ukládají ve struktuře `ipc_perm`, která obsahuje uživatelské a skupinové identifikátory procesů, které objekt vytvořily a vlastní, dále přístupový režim tohoto objektu (pro vlastníka, skupinu a ostatní) a konečně klíč IPC objektu. Tento klíč slouží k určení referenčního identifikátoru IPC objektu. Podporují se dva typy klíčů: veřejný a privátní. Pokud je klíč veřejný, může referenční identifikátor objektu zjistit kterýkoliv proces v systému, pokud k tomu má dostatečná práva. S IPC objekty se nikdy nemanipuluje pomocí klíče, vždy pouze prostřednictvím referenčního identifikátoru.

5.3.2 Fronty zpráv

Fronty zpráv umožňují jednomu nebo více procesům posílat zprávy, které jeden nebo více procesů bude číst. Linux udržuje seznam front zpráv, vektor `msgque`, jehož každý prvek ukazuje na jednu strukturu `msqid_ds`, jež plně popisuje jednu frontu zpráv. Když se vytváří fronta zpráv, alokuje se v systémové paměti nová datová struktura `msgid_ds` a přidá se do vektoru.



Obrázek 5.2
Fronty zpráv

5 Každá datová struktura `msgid_ds` obsahuje datovou strukturu `pic_perm` a ukazatele na zprávy v této frontě. Kromě toho Linux ukládá časy modifikace fronty jako například čas posledního zápisu do fronty a podobně. Struktura `msgid_ds` obsahuje dále dvě čekací fronty, jednu pro písaře fronty zpráv a druhou pro její čtenáře.

Vždy když se proces pokusí o zápis zprávy do fronty, porovnají se jeho efektivní uživatelské a skupinové ID s hodnotami v datové struktuře `ipc_perm` fronty. Pokud proces má právo zápisu do fronty, může se zpráva zkopírovat z adresového prostoru procesu do datové struktury `msg`, která se přidá na konec fronty zpráv. Může se nicméně stát, že pro zprávu nebude dostatek volného místa, protože Linux omezuje počet a délku zpráv, které je možno zapsat. V takovém případě se proces umístí do čekací fronty písařů fronty a plánovač naplánuje spuštění dalšího procesu. K probuzení písaře dojde po přečtení jedné nebo více zpráv z fronty zpráv.

Čtení z fronty je podobný proces. Nejprve se opět kontrolují přístupová práva k frontě. Čtenář si může zvolit přečtení první zprávy ve frontě bez ohledu na její typ, nebo může vybrat zprávu určitého typu. Pokud zadaným kritériím nevyhovuje žádná zpráva, čtenář bude přesunut do čekací fronty čtenářů a plánovač spustí jiný proces. Když dojde k zápisu nové zprávy do fronty, proces bude probuzen a znovu spuštěn.

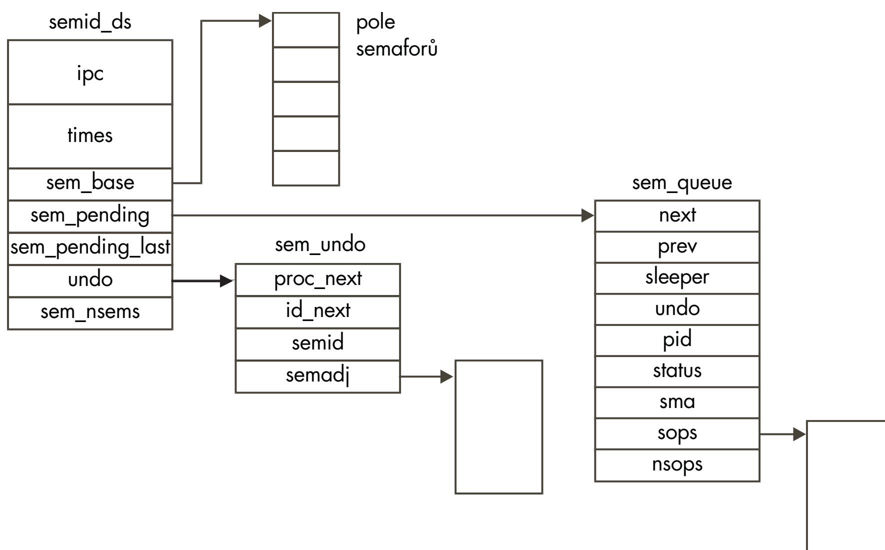
5.3.3 Semafory

Ve své nejjednodušší podobě je semafor místo v paměti, které může více procesů testovat a nastavovat. Operace testování a nastavování je z hlediska procesu nepřerušitelná, atomická - jakmile je jednou vyvolána, nic ji nemůže zastavit. Výsledkem operace testování a nastavení je součet aktuální hodnoty semaforu a nastavované hodnoty, která může být kladná nebo záporná. Podle výsledku operace může být proces uspán dokud hodnota semaforu nebude změněna jiným procesem. Semafory se dají použít k implementaci *kritických sekcí*, kritických oblastí kódu, které může v jednom okamžiku vykonávat pouze jediný proces.

Řekněme, že máte mnoho spolupracujících procesů, které všechny čtou a zapisují záznamy v jednom datovém souboru. Budete zřejmě potřebovat přísně omezit přístup k souboru. Můžete použít semafor s počáteční hodnotou jedna a kolem kódu pro operaci se souborem umístit dvě semaforové operace: první bude testovat a dekrementovat hodnotu semaforu, druhá ji bude testovat a inkrementovat. První proces, který se pokusí o přístup k souboru, bude dekrementovat semafor a zdaří-li se mu to, bude mít semafor nově hodnotu 0. Tento proces nyní může pokračovat dále a používat datový soubor, pokud se však nyní pokusí o dekrementaci semaforu jiný proces, operace se nezdaří protože nová hodnota semaforu by byla -1. Druhý proces bude pozastaven do doby, než první proces inkrementuje hodnotu semaforu zpět na 1. Nyní je možno pozastavený proces probudit a tentokrát pokus o dekrementaci semaforu uspěje.

Každý objekt semaforu je popsán polem, Linux k tomuto účelu používá datovou strukturu `semid_ds`. Na všechny datové struktury `semid_ds` v systému se odkazuje pole `semary`, vektor ukazatelů. Všechny procesy, které mohou manipulovat s polem určitého objektu semafor, s polem manipulují pomocí systémových volání. Systémové volání může specifikovat mnoho operací, přičemž každá je popsána třemi vstupními hodnotami: indexem semaforu, operační hodnotou a množinou příznaků. Index semaforu je index do pole semaforů, ope-

rační hodnota je číslo, které se přičte k aktuální hodnotě semaforu. Nejprve Linux testuje, zda operace může proběhnout úspěšně. Operace proběhne úspěšně v případě, že po přičtení operační hodnoty k aktuální hodnotě semaforu bude výsledek větší než nula, nebo pokud jsou nulové jak operační hodnota, tak aktuální hodnota. Pokud by operace proběhla neúspěšně, Linux pozastaví proces, ovšem pouze v případě, že nebyl aktivní příznak neblokující operace. Pokud proces bude pozastaven, Linux musí uložit stav požadované operace a poté proces přemístit do čekací fronty. Dosáhne toho vytvořením datové struktury `sem_queue` na zásobníku a jejím naplněním hodnotami předávanými při volání operace. Nová datová struktura `sem_queue` se umístí na konec čekací fronty semaforu (pomocí ukazatelů `sem_pending` a `sem_pending_last`). Aktuální proces se umístí do čekací fronty v datové struktuře `sem_queue` (položka `sleeper`) a volá se plánovač, který spustí další proces.



Obrázek 5.3
Semafory

Pokud by všechny požadované semaforové operace uspěly a proces není nutno pozastavit, Linux pokračuje a provede požadované operace nad požadovanými položkami pole semaforů. Dále musí Linux otestovat, zda některý z čekajících procesů může nyní uspět ve své požadované operaci. Projde všechny položky fronty čekajících procesů (`sem_turn`) a testuje, zda jejich operace tentokrát uspěje. Pokud ano, odstraní datovou strukturu `sem_queue` z fronty čekajících procesů a provede nad polem semaforů požadovanou operaci. Probudí čekající proces, takže jej bude možno spustit při příštím chodu plánovače. Linux zopakuje průchod frontou čekajících procesů znovu od začátku a hledá, zda není možno obsloužit další požadavek až do chvíle, kdy už není možno žádnou operaci provést a není možno probudit žádný proces.

U semaforu se objevuje nebezpečí *zablokování - deadlock*. Dojde k tomu, pokud by nějaký proces při vstupu do kritické sekce změnil stav semaforu, avšak poté by kritickou sekci korektně neopustil například protože havaruje nebo je explicitně zrušen. Linux se proti zablokování chrání udržováním seznamu změn pole semaforů. Smyslem je, aby se podle seznamu změn dal semafor uvést do stejného stavu, v jakém byl před provedením operací požadovaných zaniklým procesem. Změny se ukládají v datových strukturách `sem_undo` ukládaných ve frontě ve strukturách `semid_ds` i `task_struct` těch procesů, které používají semafory.

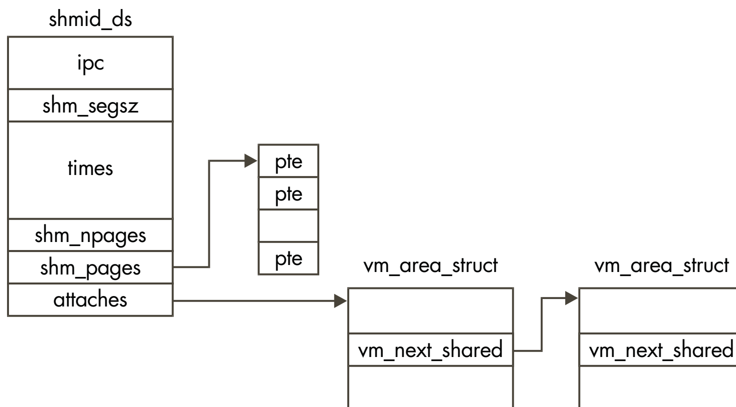
Každá jednotlivá semaforová operace může žádat o zaznamenání změny. Linux pro každý proces a každé semaforové pole spravuje minimálně jednu strukturu `sem_undo`. Pokud proces žádnou nemá, Linux ji v okamžiku potřeby vytvoří. Nová struktura `sem_undo` se zařadí do front jak ve struktuře `task_struct` procesu, tak ve struktuře `semid_ds` semaforu. Operace prováděné nad polem semaforů se negují a zapisují do příslušných položek pole změn v této struktuře `sem_undo`. Pokud je tedy operační hodnota požadavku 2, do pole změn se přičte hodnota -2.

Když dojde k zániku procesu, Linux při jeho ukončení prochází datové struktury `sem_undo` a provádí změny na příslušných semaforových polích. Pokud daný semafor neexistuje, zůstává struktura `sem_undo` zařazena ve frontě struktury `task_struct`, identifikátor semaforu však není platný. V takovém případě úklidový kód semaforu datovou strukturu `sem_undo` jednoduše zruší.

5.3.4 Sdílená paměť

Sdílená paměť umožňuje jednomu nebo více procesům komunikovat prostřednictvím oblasti paměti, která se objevuje ve virtuálním adresovém prostoru každého z nich. Na stránky virtuální paměti je uveden odkaz prostřednictvím položek v tabulce stránek každého ze sdílejících procesů. Sdílená paměť se u jednotlivých procesů nemusí objevovat na stejném místě virtuální paměti. Stejně jako u všech IPC objektů Systemu V je i přístup k oblastem sdílené paměti řízen pomocí klíčů a kontroly přístupových práv. Jakmile je sdílení paměti povoleno, nijak už se nekontroluje způsob využití sdílené paměti jednotlivými sdílejícími procesy. Ty musí při synchronizaci přístupu do sdílené paměti používat jiné mechanismy, například semafory.

Každá nově vytvořená oblast sdílené paměti je reprezentována datovou strukturou `shmid_ds`. Tyto struktury se ukládají ve vektoru `shm_segs`. Datová struktura `shmid_ds` popisuje jak velká je oblast sdílené paměti, kolik procesů ji používá a jak se sdílená paměť mapuje do jejich adresových prostorů. Přístupová práva k paměti a typ klíče je určen tím, kdo sdílenou paměť vytvořil. Pokud má tvůrce sdílené oblasti paměti dostatečná práva, může dokonce nařídit trvalé uzamčení sdílené oblasti ve fyzické paměti.



Obrázek 5.4
Sdílená paměť

Každý proces, který si přeje paměť sdílet, se k ní musí připojit pomocí systémového volání. To vytvoří novou datovou strukturu vm_area_struct popisující sdílenou paměť v tomto procesu. Proces může sám rozhodnout, kam v jeho adresovém prostoru se má sdílená paměť mapovat, nebo může toto rozhodnutí ponechat na operačním systému. Nová struktura vm_area_struct se připojí do seznamu těchto struktur, na něž ukazuje struktura shmid_ds. K propojení struktur týkajících se oblastí sdílené paměti slouží ukazatele vm_next_shared a vm_prev_shared. V průběhu připojení se virtuální paměť fakticky nevytváří, k tomu dojde až v okamžiku prvního přístupu ke sdílené oblasti.

Když se proces poprvé pokusí o přístup do některé ze stránek sdílené paměti, dojde k výpadku stránky. Při obsluze výpadku Linux nejprve hledá strukturu vm_area_struct, která popisuje vypadnuvší stránku. Ta obsahuje ukazatele na obslužné rutiny příslušného typu virtuální paměti. Obslužný kód výpadku sdílené stránky hledá v položkách tabulky stránek strukturu shmid_ds a zjišťuje, zda pro danou stránku struktura existuje. Pokud neexistuje, provede alokaci fyzické stránky a vytvoří pro ni položku v tabulce stránek. Kromě tabulky stránek aktuálního procesu se položka umístí i ve struktuře shmid_ds. Znamená to, že když se o přístup ke stejné oblasti sdílené paměti pokusí další proces a dojde k výpadku stránky, použije obslužný kód výpadku pro tento proces stejnou již vytvořenou fyzickou stránku. Tedy první proces přistupující ke sdílené oblasti paměti způsobí vytvoření sdílené paměti a u všech dalších procesů už dochází pouze k namapování fyzicky existující sdílené paměti do jejich virtuálního adresového prostoru.

Jakmile proces nebude sdílenou paměť dále potřebovat, odpojí se od ní. Dokud existují jiné procesy, které stejnou sdílenou oblast ještě využívají, je odpojení záležitostí pouze aktuálního procesu. Z datové struktury shmid_ds se odstraní jeho položka vm_area_struct a zruší se. Aktualizuje se tabulka stránek aktuálního procesu a oblast paměti po-

užívaná pro sdílení se označí za neplatnou. Když se od sdílené paměti odpojí poslední proces, dojde k uvolnění stránek sdílené paměti z fyzické paměti a zruší se také struktura `shmid_ds` dané oblasti sdílené paměti.

Další komplikace při práci se sdílenou pamětí se objevují v okamžiku, pokud stránky sdílené paměti nejsou uzamčeny ve fyzické paměti. V takovém případě může dojít k odložení sdílené paměti na disk v době, kdy je nedostatek fyzické paměti. Mechanismus odkládání sdílených paměťových stránek je popsán v kapitole Správa paměti.

Odkazy na zdrojové texty jádra

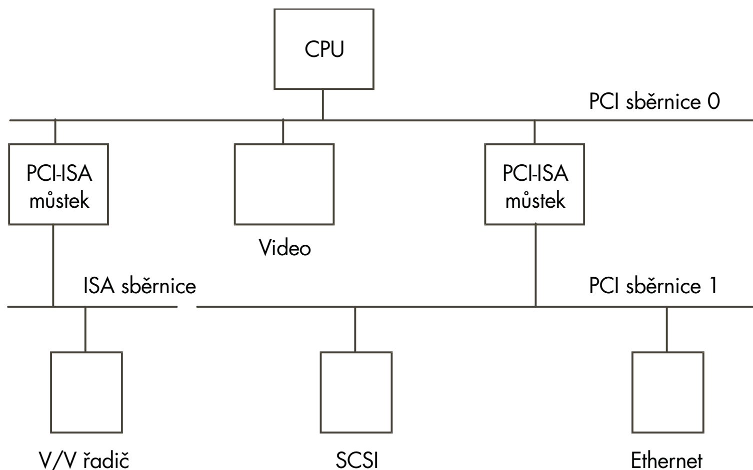
1 – `Viz include/linux/-inode_fs_i.h`
2 – `Viz pipe_write()in fs/pipe.c`
3 – `Viz pipe_read()in fs/pipe.c`
4 – `Viz include/-linux/ipc.h`
5 – `Viz include/-linux/msg.h`
6 – `Viz include/-linux/sem.h`
7 – `Viz include/-linux/sem.h`

Kapitola

6

PCI

Peripheral Component Interconnect (PCI)* je, jak už jeho jméno napovídá, standard popisují způsob propojování periferních zařízení počítačového systému strukturovaným a řízeným způsobem. Standard popisuje jednak způsoby elektrického připojení jednotlivých komponent a také způsoby, jakými se mají zařízení chovat. V této kapitole hovoříme o metodách, jimiž jádro Linuxu inicializuje sběrnice a zařízení PCI.



Obrázek 6.1
Příklad PCI systému

* Poznámka korektora: Celá kniha je zaměřena na jádra verze 2.0.x, v jádrech 2.1.x je celý systém PCI díky Martinu Marešovi kompletně přeprogramován.

Na obrázku 6.1 vidíme logický diagram systému PCI. sběrnice PCI a můstky PCI-PCI představují pojivo, které vzájemně propojuje komponenty systému. Procesor je připojen na sběrnici PCI 0, primární sběrnici, PCI stejně jako videokarta. Speciální zařízení PCI, můstek PCI-PCI, propojuje primární sběrnici PCI se sekundární sběrnicí PCI, sběrnicí PCI 1. V terminologii specifikace PCI se sběrnice PCI 1 označuje jako *downstream* můstku PCI-PCI, sběrnice 0 se označuje jako *upstream* můstku. K sekundární sběrnici PCI jsou připojeny SCSI řadič a ethernetová karta. Fyzicky mohou být jak můstek, sekundární sběrnice i obě zařízení součástí jedné karty PCI. můstek PCI-ISA slouží pro podporu starších zařízení ISA, na obrázku máme jako příklad zařízení ISA uveden kombinovaný řadič, sloužící k připojení řekněme myši a diskety.

6.1 Adresové prostory PCI

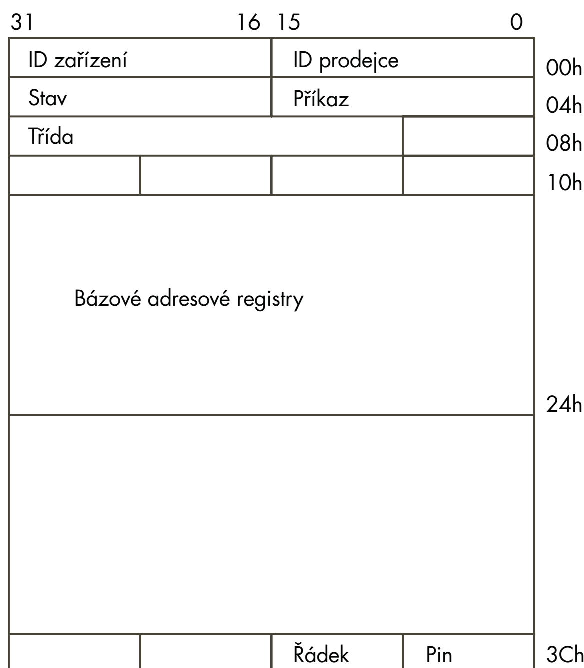
Procesor a zařízení PCI potřebují přístup ke vzájemně sdílené oblasti paměti. Tuto paměť používají ovladače zařízení k řízení zařízení PCI a k výměně informací mezi sebou. Typicky sdílená paměť obsahuje řídicí a stavové registry zařízení. Tyto registry slouží k řízení zařízení a ke čtení jeho stavu. Například ovladač PCI řadiče SCSI bude číst obsah stavového registru při zjišťování, zda je zařízení SCSI připraveno zapsat blok informací na disk SCSI. Zápisem do řídícího registru může provést aktivaci zařízení po jeho zapnutí.

Jako sdílenou paměť by bylo možno využít systémovou paměť procesoru, pokud by tomu ale tak bylo, vždy když by zařízení přistupovalo k paměti, musel by být procesor pozastaven a čekat na dokončení přístupu zařízením. Přístup k paměti je obecně omezen vždy jen na jednu komponentu v systému. Tímto způsobem by tedy došlo ke zpomalení systému. Rovněž není příliš vhodné, aby periferní zařízení mohla libovolným způsobem manipulovat s hlavní operační pamětí. Bylo by to velmi nebezpečné, protože nevhodnými manipulacemi by systém mohl být uveden do zcela nestabilního stavu.

Periferní zařízení mají svůj vlastní adresový prostor. Procesor k němu může přistupovat, avšak přístup zařízení k systémové paměti je přísně řízen prostřednictvím kanálů DMA. Zařízení ISA mají přístup ke dvěma adresovým prostorům: ISA V/V a paměti ISA. Zařízení PCI používají tři oblasti: PCI V/V, paměť PCI a konfigurační prostor PCI. Všechny tyto oblasti jsou přístupné také procesoru s tím, že PCI V/V a paměť PCI používají ovladače zařízení, konfigurační prostor PCI je využíván jádrem při inicializaci PCI.

Procesor Alpha AXP neumí přímo přistupovat k jiným adresovým oblastem než je systémový adresový prostor. Používá proto podpůrné čipy, které zajišťují přístup do jiných adresových prostorů, jako je například konfigurační prostor PCI. Používá rozptýlené mapovací schéma, které „ukradne“ část obrovského virtuálního adresového prostoru a mapuje do ní adresový prostor zařízení PCI.

6.2 Konfigurační hlavičky PCI



Obrázek 6.2
Konfigurační hlavička PCI

Každé zařízení PCI v systému včetně můstků PCI-PCI používá konfigurační datovou strukturu, která je umístěna někde v konfiguračním adresovém prostoru PCI. Konfigurační hlavička slouží systému k identifikaci a řízení zařízení. Přesné umístění hlavičky v konfiguračním prostoru záleží na umístění zařízení v topologii PCI. Například videokarta PCI zapojená v určitém slotu PCI na základní desce bude mít konfigurační hlavičku na určitém místě prostoru, zapojíme-li ji do jiného slotu, pak se hlavička objeví v jiném místě konfiguračního prostoru. To ovšem nevadí, protože ať je zařízení nebo můstek umístěno kdekoliv, systém je vždy najde a nakonfiguruje pomocí stavových a konfiguračních registrů v hlavičce.

Typicky bývají systémy navrženy tak, aby každý slot PCI měl svou konfigurační hlavičku na offsetu, který odpovídá pozici slotu na desce. Takže například první slot na desce bude mít svou hlavičku na offsetu 0 konfiguračního prostoru, druhý slot na offsetu 256 (všechny hlavičky mají pevnou délku 256 bajtů) a tak dále. Musí existovat systémově závislý hardwarový mechanismus, který umožní konfiguračnímu kódu prozkoumat všechny možné konfigurační hlavičky sběrnice PCI a zjistit, která zařízení jsou připojena a která ne, a to prostým čtením jednoho pole každé hlavičky (typicky položky *Identifikátor výrobce*) a generováním nějakých

chyb. Standard definuje jedno možné chybové hlášení jako vrácení hodnoty *0xFFFFFFFF* při pokusu o čtení *Identifikátoru výrobce* nebo *Identifikátoru zařízení*, čímž se indikuje nevyužitý slot PCI.

1 Na obrázku 6.2 je znázorněna struktura konfigurační hlavičky. Skládá se z následujících polí:

Identifikátor výrobce Jednoznačné číslo popisující výrobce zařízení PCI. Identifikátor zařízení PCI firmy Digital je *0x1011*, Intel používá identifikátor *0x8086*.

Identifikátor zařízení Jednoznačné číslo popisující samotné zařízení. Například rychlá etherentová karta 21141 firmy Digital má identifikátor *0x0009*.

Status Toto pole obsahuje status zařízení s tím, že význam jednotlivých bitů pole je definován standardem.

Příkaz Zápisem do tohoto pole systém řídí zařízení, například povoluje zařízení přístup do PCI V/V adresového prostoru.

Kód třídy Identifikátor typu zařízení. Pro každý typ zařízení, například video, SCSI a podobně, existuje standardní třída zařízení. Zařízení SCSI mají přidělenu třídu *0x0100*.

Registry bázové adresy Tyto registry slouží k určení a alokaci typu, velikosti a umístění adresového prostoru PCI V/V a paměti PCI, které zařízení může používat.

Přerušovací pin Přerušení od karty se na sběrnici PCI předávají prostřednictvím čtyř fyzických pinů. Standard se označují jako piny A, B, C a D. Položka *přerušovací pin* udává, který z těchto pinů zařízení PCI používá. Obecně je tento údaj pevně dán při výrobě zařízení. Znamená to, že při každém spuštění používá zařízení stejný přerušovací pin. Tato informace slouží subsystému obsluhy přerušení k obsluze přerušení od zařízení.

Přerušovací linka Pole *přerušovací linka* slouží k předání handlu přerušení mezi inicializačním kódem PCI, ovladačem zařízení a subsystémem obsluhy přerušení. Zde zapsaná hodnota nemá pro ovladač zařízení význam, umožňuje však obsluze přerušení správně směrovat přerušení od zařízení PCI na obslužný kód přerušení správného ovladače zařízení v systému Linux. Podrobnosti o obsluze přerušení v Linuxu jsou uvedeny v kapitole „Přerušení a jejich obsluha“.

6.3 Adresy PCI V/V a paměti PCI

Tyto dva adresové prostory slouží zařízením pro komunikaci s jejich ovladačem, který běží na procesoru v jádře Linuxu. Například rychlá ethernetová karta DEC 21141 mapuje své interní registry právě do prostoru PCI V/V. Její ovladač v jádře Linuxu může prostřednictvím čtení a zápisu do těchto registrů kartu řídit. Videokarty typicky používají velké oblasti paměti PCI k ukládání videoinformací.

Dokud nedojde k nastavení systému PCI a není povolen přístup zařízení do těchto adresových prostorů pomocí pole *Příkaz* v konfigurační hlavičce, žádné zařízení nesmí těchto adresových prostorů využívat. Je třeba zdůraznit, že konfigurační prostor PCI je přístupný pouze konfiguračnímu kódu v jádře, ovladače zařízení pak pracují pouze s prostorem V/V a paměťovým prostorem.

6.4 Můstky PCI-ISA

Tyto můstky slouží k podpoře starších zařízení ISA. Překládají přístupy do adresových prostorů PCI na přístup do adresových prostorů zařízení ISA. V dnešní době je většina systémů vybavena několika sloty ISA a několika sloty PCI. Postupem času potřeba podpory starších zařízení poklesne a budou k dispozici už pouze systémy PCI. Umístění jednotlivých zařízení v adresovém prostoru ISA (ISA V/V a ISA paměti) bylo zavedeno v temném dávnověku prvních PC s procesorem Intel 8080. Dokonce i nejmodernější počítač s procesorem Alpha AXP za pět tisíc dolarů má řadič disketových mechanik ISA mapován na stejných adresách ISA jako první počítače IBM PC. Specifikace PCI se s tímto problémem vyrovnává tak, že nižší oblasti adresových prostorů PCI rezervuje pro periferie ISA a používá jednoduchý můstek PCI-ISA, který překládá přístupy do paměti PCI na přístup do odpovídajících oblastí paměti ISA.

6.5 Můstky PCI-PCI

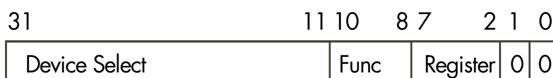
Můstky PCI-PCI jsou speciální zařízení PCI, která sdružují dohromady sběrnice PCI v systému. Jednoduché systémy mají jedinou sběrnici PCI, ovšem počet zařízení, které může jedna sběrnice PCI podporovat, je omezen jejími elektrickými vlastnostmi. Když se pomocí můstů PCI-PCI přidají další sběrnice PCI, může systém podporovat více zařízení PCI. To je důležité zejména u vysoce výkonných serverů. Linux samozřejmě plně podporuje funkci můstků PCI-PCI.

6.5.1 Můstky PCI-PCI: okna PCI V/V a paměti PCI

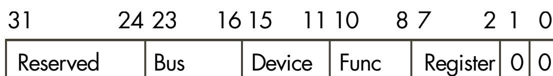
Můstky PCI-PCI předávají směrem dolů pouze vybranou podmnožinu požadavků na PCI V/V a paměťový prostor PCI. Například na obrázku 6.1 bude můstek PCI-PCI předávat ze sběrnice PCI 0 na sběrnici PCI 1 pouze ty požadavky na čtení a zápis, které se vztahují na oblasti V/V a paměti přiřazené buď SCSI nebo ethernetovému zařízení, ostatní požadavky bude ignorovat. Tato filtrace zamezuje zbytečnému šíření adres v systému. Aby mohl můstek takovouto filtrační funkci plnit, musí mít naprogramovány bázovou adresu a limit V/V prostoru a paměťového prostoru, které má předávat ze své primární sběrnice na svou sekundární sběrnici. Jakmile je jednou můstek naprogramován, stává se neviditelným, protože ovladače zařízení přistupují ke svým zařízením pouze prostřednictvím přidělených oken adresových prostorů. To je důležitá funkce, která usnadňuje práci autorům ovladačů zařízení PCI. Na druhé straně se tím ale pro jádro poněkud komplikuje způsob konfigurace můstku, jak uvidíme dále.

6.5.2 Můstky PCI-PCI: konfigurační cykly a číslování PCI sběrnic

Aby mohl inicializační kód PCI adresovat zařízení, která nejsou připojena na hlavní sběrnici PCI, musí existovat mechanismus, který můstkům umožní rozhodnout, zda předat konfigurační cykly z primárního na sekundární rozhraní. Cyklus je vlastně adresa, která se objevuje na sběrnici PCI. Specifikace PCI definuje dva formáty konfiguračních adres PCI, typ 0 a typ 1, které jsou znázorněny na obrázcích 6.3 a 6.4. Konfigurační cyklus typu 0 neobsahuje číslo sběrnice a všechna zařízení jej chápou jako konfigurační cyklus zařízení na této sběrnici. Bity 31 až 11 konfiguračního cyklu 0 se chápou jako pole výběru zařízení. Jedna možnost při návrhu systému je každým bitem vybírat jedno zařízení. V takovém případě by se bitem 11 vybíralo zařízení na PCI slotu 0, bitem 12 zařízení na slotu 1 a tak dále. Druhá možnost je na bity 31 až 11 zapisovat přímo čísla slotů PCI. Použitá metoda závisí na řadiči PCI paměti.



Obrázek 6.3
Konfigurační cyklus 0



Obrázek 6.4
Konfigurační cyklus 1

Konfigurační cyklus typu 1 obsahuje číslo sběrnice PCI a tento typ cyklu je ignorován všemi zařízeními s výjimkou můstků PCI-PCI. Když můstek PCI-PCI uvidí cyklus typu 1, rozhodne se, zda jej předat sekundární sběrnici. Zda můstek bude cyklus typu 1 ignorovat nebo zda jej předá sekundární sběrnici závisí na jeho konfiguraci. Každý můstek PCI-PCI má přiděleno číslo primární sběrnice a číslo sekundární sběrnice. Primární sběrnice je sběrnice blíže procesoru, sekundární sběrnice je ta dále od procesoru. Každý PCI-PCI můstek dále zná číslo podřízené PCI sběrnice, což je nejvyšší číslo ze všech sběrnic PCI, které jsou připojeny dalšími můstky k sekundární sběrnici tohoto můstku. Jinak řečeno, číslo podřízené sběrnice je nejvyšší číslo PCI sběrnice směrem dolů od můstku. Když můstek PCI-PCI zaregistruje konfigurační cyklus typu 1, provede jednu ze tří následujících operací:

- Ignoruje cyklus pokud číslo sběrnice uvedené v cyklu neleží mezi číslem sekundární sběrnice můstku a číslem podřízené sběrnice (včetně).
- Konvertuje cyklus na typ 0 pokud číslo sběrnice v cyklu odpovídá číslu sekundární sběrnice můstku.
- Předává nezměněný cyklus na sekundární rozhraní, pokud číslo sběrnice je větší než číslo sekundární sběrnice, ale menší nebo rovno číslu podřízené sběrnice.

Pokud tedy budeme chtít adresovat zařízení 1 na sběrnici 3 podle topologie uvedené na obrázku 6.9, bude muset procesor generovat konfigurační příkaz typu 1. Můstek 1 jej nezměněný předá můstku 2. Můstek 2 jej bude ignorovat, ale můstek 3 jej zkonvertuje na konfigurační příkaz typu 0 a předá jej na sběrnici 3, kde na něj zareaguje zařízení 1.

Mechanismus alokace čísel jednotlivým sběrnicím PCI při inicializaci systému je plně v moci příslušného operačního systému, pro všechny můstky PCI-PCI však musí platit následující pravidlo:

Čísla všech sběrnic pod můstkem PCI-PCI musí být větší než číslo sekundární sběrnice a menší nebo rovna číslu podřízené sběrnice.

Pokud by toto pravidlo bylo porušeno, můstky PCI-PCI by neprováděly správně překlad a předávání konfiguračních cyklů typu 1 a systému by se nepodařilo nalézt a inicializovat zařízení PCI v systému. Kvůli dodržení číslovacího schématu konfiguruje Linux tato speciální zařízení v určitém pořadí. V dále uvedené části „Přiřazení čísel sběrnic“ je podrobněji popsáno schéma číslování sběrnic PCI.

6.6 Inicializace PCI v Linuxu

Inicializační kód PCI je v Linuxu rozdělen na tři logické části:

Ovladač zařízení PCI Tento pseudoovladač zařízení prohledává systém PCI počínaje sběrnicí 0 a nalezne všechna zařízení PCI a můstky v systému. Vytvoří seznam datových struktur popisujících topologii systému. Navíc očísluje všechny nalezené můstky.

2

PCI BIOS Tato softwarová vrstva zajišťuje služby popsané ve specifikaci PCI BIOSu. Přestože systémy Alpha přímo neposkytují služby BIOSu, v jádře Linuxu je obsažen ekvivalentní kód, zajišťující stejné funkce.

3

4

PCI fixup Systémově specifický kód zajišťuje systémově závislé dokončení inicializace PCI.

6.6.1 Datové struktury PCI v jádře

Když jádro Linuxu inicializuje systém PCI, buduje si datové struktury, které zrcadlí skutečnou topologii systému PCI. Na obrázku 6.5 je vidět vztah těchto datových struktur, které by odpovídaly struktuře PCI na obrázku 6.1.

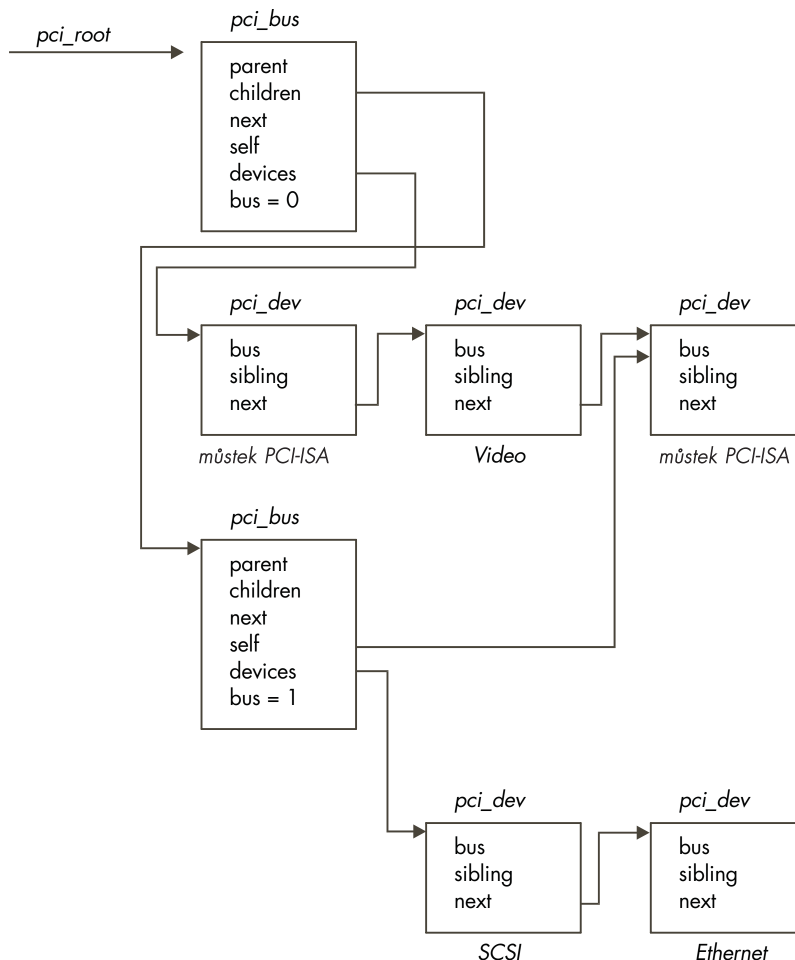
Každé zařízení PCI (včetně můstků PCI-PCI) je popsáno datovou strukturou `pci_dev`. Každá sběrnice PCI je popsána strukturou `pci_bus`. Výsledkem je stromová struktura PCI sběrnic, která má každá k sobě připojeno několik zařízení PCI. Protože sběrnice PCI je dosažitelná pouze prostřednictvím můstku PCI-PCI (s výjimkou primární sběrnice, PCI 0), obsahuje každá struktura `pci_bus` ukazatel na zařízení PCI (můstek PCI-PCI), které ji zpřístupňuje. Toto zařízení je synovským zařízením rodičovské sběrnice té sběrnice, k níž je uvažovaná sekundární sběrnice připojena.

Na obrázku 6.5 není znázorněn ukazatel na všechna zařízení PCI v systému, ukazatel `pci_devices`. Každé zařízení PCI v systému má svou strukturu `pci_dev` zařazenu v seznamu uvedeným tímto ukazatelem. Tento seznam slouží jádru k rychlému nalezení všech zařízení PCI v systému.

6.6.2 Ovladač zařízení PCI

Ovladač zařízení PCI není ve skutečnosti vůbec ovladačem zařízení, jedná se o funkci operačního systému, která se volá v době inicializace systému. Inicializační kód PCI musí prohlédnout všechny sběrnice PCI v systému a nalézt všechna zařízení PCI včetně můstků PCI-PCI.

5



Obrázek 6.5
Datové struktury PCI v jádře

Používá kód PCI BIOSu k nalezení všech možných slotů na právě prohledávané sběrnici PCI. Pokud je slot obsazen, vytvoří strukturu `pci_dev` popisující zařízení a připojí ji k seznamu známých zařízení PCI (na který ukazuje `pci_devices`).

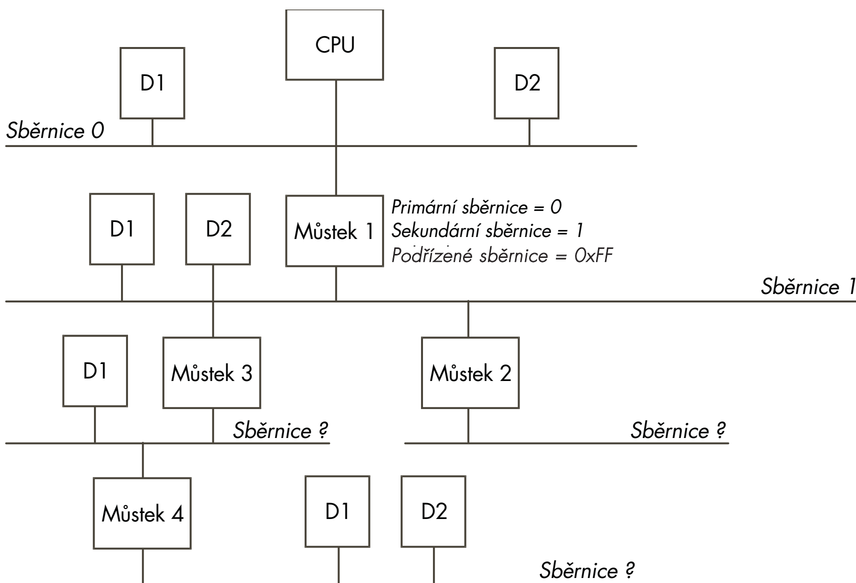
Inicializační kód PCI začíná od sběrnice PCI 0. Pokouší se načíst *identifikátor výrobce* a *identifikátor zařízení* každého možného zařízení ve všech slotech. Když nalezne obsazený slot, vytvoří strukturu `pci_dev` popisující zařízení. Všechny struktury `pci_dev` vytvořené inicializačním kódem PCI (včetně struktur pro můstky PCI) jsou zařazeny do jednosměrně propojeného seznamu `pci_devices`.

Pokud je nalezené zařízení PCI můstkem, vytvoří se struktura `pci_bus` a připojí se ke stromu struktur `pci_bus` a `pci_dev`, na kterou ukazuje `pci_root`. Inicializační kód je schopen rozpoznat můstky, protože všechny můstky mají přidělen kód třídy *0x060400*. Poté jádro Linuxu nakonfiguruje sběrnici PCI na druhé (sekundární) straně právě nalezeného můstku. Pokud je nalezeno více můstků PCI-PCI, nakonfigurují se všechny. Tento proces se označuje jako algoritmus „do hloubky“, protože sběrnice je nejprve zmapována ve směru „do hloubky“ a teprve pak se mapuje „do šířky“. Když se budeme držet obrázku 6.1, nakonfiguruje Linux sběrnici 1 a její SCSI a etherentové zařízení dříve, než provede konfiguraci videozařízení na sběrnici 0.

Když Linux vyhledává sběrnice PCI, musí rovněž konfigurovat mezilehlé můstky PCI-PCI a přidělit jim čísla sekundární a podřízené sběrnice. Tento postup je podrobně popsán v následující části.

Konfigurace můstků PCI-PCI – Přidělování čísel sběrnic

Aby můstek mohl propouštět zápisy a čtení ve V/V prostoru, paměťovém prostoru a konfiguračním prostoru, potřebuje znát následující informace:



Obrázek 6.6
Konfigurace systému PCI, část 1

číslo primární sběrnice Číslo sběrnice bezprostředně nadřazené můstku.

číslo sekundární sběrnice Číslo sběrnice bezprostředně podřízené můstku.

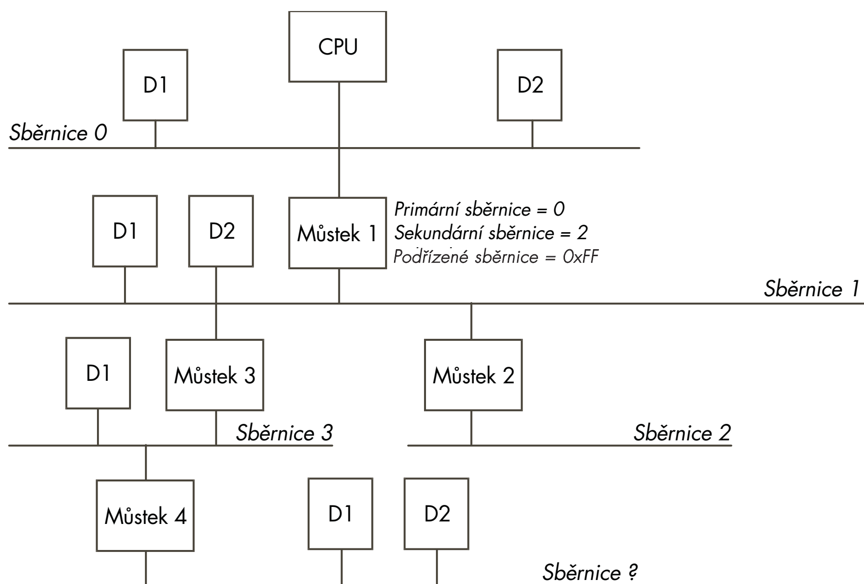
číslo podřízené sběrnice Nejvyšší ze všech čísel sběrnic, které jsou přes můstek směrem dolů dosažitelné.

okna V/V a paměťové oblasti Bázová adresa a velikost V/V prostoru a paměťového prostoru pro všechny směrem dolů adresovatelné sběrnice.

Problém je v tom, že v době kdy chcete nakonfigurovat nějaký můstek PCI-PCI, neznáte číslo jeho podřízené sběrnice. Nevíte, zda je pod ním další můstek a i kdybyste to věděli, nevíte, jaká čísla budou přiřazena pod ním. Odpovědí je použití hloubkového rekurzivního algoritmu, který hledá můstky na všech sběrnicích a jak je nachází, přiřazuje jim čísla. Když je nalezen můstek a očísluje se jeho sekundární sběrnice, jako číslo podřízené sběrnice se dočasně přiřadí číslo *0xFF* a prohledávají se a číslují všechny můstky a sběrnice pod ním. Celé to vypadá složitě, avšak následující příklad by měl vše vyjasnit.

Číslování můstků PCI-PCI – krok 1

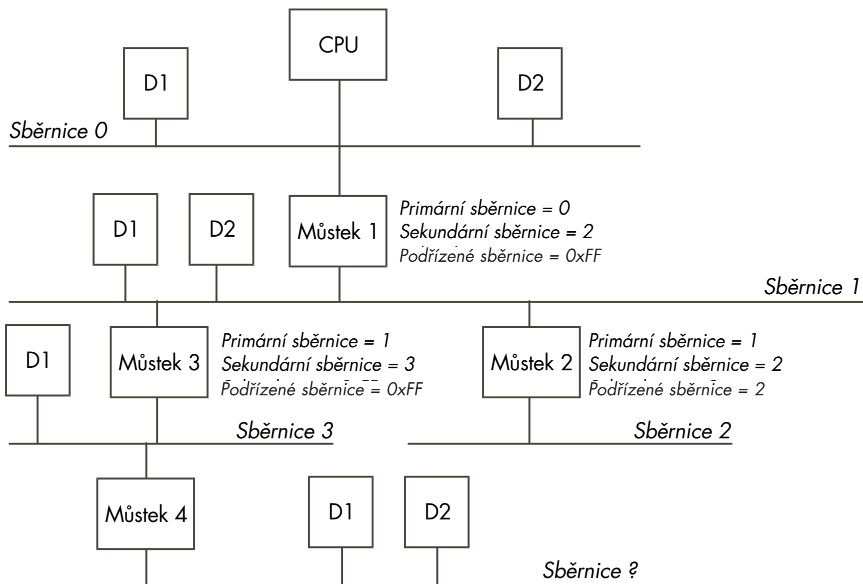
Když vezmeme topologii podle obrázku 6.6, nalezne se jako první můstek 1. Sběrnice PCI pod tímto můstkem bude očíslována jako 1 a můstku 1 bude přiřazeno číslo sekundární sběrnice rovno jedné a číslo podřízené sběrnice rovno dočasně *0xFF*. Znamená to, že všechny konfigurační cykly typu 1 určující sběrnici PCI číslo 1 a vyšší projdou můstkem 1 na sběrnici 1. Pokud mají číslo sběrnice rovno 1, budou tímto můstkem přeloženy na cykly typu 0, pokud mají vyšší číslo sběrnice, zůstanou nezměněny. A to je přesně to, co inicializační kód potřebuje, aby mohl pokračovat a prohlédnout sběrnici 1.



Obrázek 6.7
Konfigurace systému PCI – část 2

Číslování PCI-PCI můstků - krok 2

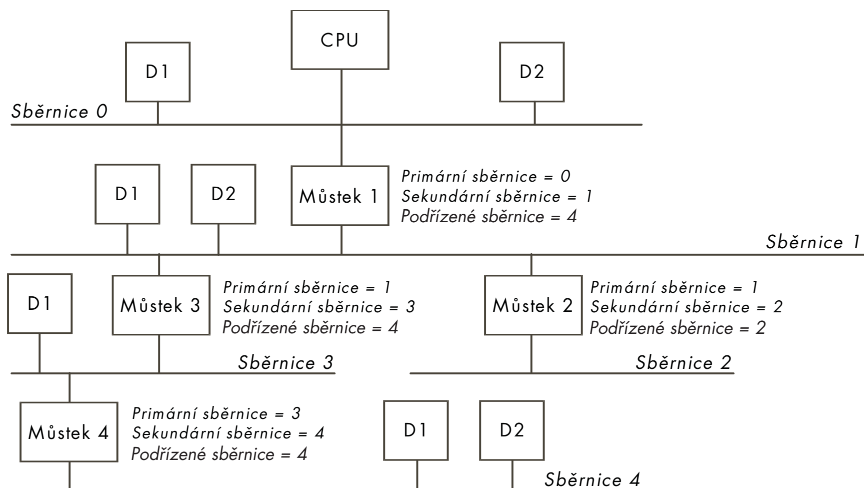
Linux používá hloubkový algoritmus, takže inicializační kód nyní začíná prohlížet můstek 1. Pod ním nalezne můstek 2. Pod můstkem 2 už nejsou žádné další můstky, takže se mu jako číslo podřízené sběrnice přiřadí 2, což odpovídá číslu jeho sekundární sběrnice. Na obrázku 6.7 je vidět přiřazení čísel sběrnic a konfigurace můstků v této fázi.



Obrázek 6.8
Konfigurace systému PCI – část 3

Číslování můstků PCI-PCI – krok 3

Inicializační kód se vrátí ke sběrnici 1 a na ní nalezne další můstek, můstek 3. Jako číslo primární sběrnice bude mít přiřazeno 1, jako číslo sekundární sběrnice 3 a jako číslo podřízené sběrnice *0xFF*. Na obrázku 6.8 vidíme, jak je systém nakonfigurován v tomto okamžiku. Konfigurační cykly typu 1 s čísly sběrnice 1, 2 a 3 se budou správně doručovat na příslušné sběrnice.



Obrázek 6.9
Konfigurace systému PCI – část 4

Číslování můstků PCI-PCI – krok 4

Linux začne prohlížet sběrnici PCI 3 pod můstkem 3. Na této sběrnici je další můstek (můstek 4), kterému se jako primární sběrnice přiřadí 3, jako sekundární sběrnice 4. Jedná se o poslední můstek v této větvi, takže jako číslo podřízené sběrnice bude mít rovněž přiřazeno 4. Inicializační kód se pak vrátí k můstku 3 a jako podřízenou sběrnici nastaví sběrnici 4. Nakonec inicializační kód přiřadí podřízenou sběrnici 4 i můstku 1. Na obrázku 6.9 vidíme finální přiřazení čísel sběrnic.

6.6.3 Funkce PCI BIOSu

Funkce PCI BIOSu jsou standardní množinou operací, které jsou společné pro všechny platformy. Jsou například stejné jak na systémech Intel, tak i Alpha AXP. Umožňují procesoru řídit přístup ke všem adresovým prostorům sběrnice PCI.

Tyto funkce může používat pouze kód jádra a ovladače zařízení.

6.6.4 PCI Fixup

Fixup kód procesoru Alpha AXP toho dělá podstatně více než stejný kód pro procesor Intel (který v zásadě nedělá vůbec nic).

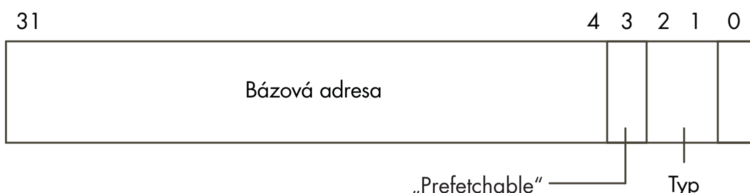
U systémů na platformě Intel provede kompletní inicializaci PCI systému BIOS, spouštěný v době startu počítače. Linuxu už toho kromě mapování konfigurace mnoho nezbývá. U jiných systémů je v další fázi konfigurace nutné provést následující operace:

- Přiřazení V/V prostoru a paměťového prostoru jednotlivým zařízením.
- Konfigurace adresových oken jednotlivých můstků.
- Generování hodnot přerušovací linky jednotlivých zařízení, které slouží k obsluze přerušení od zařízení.

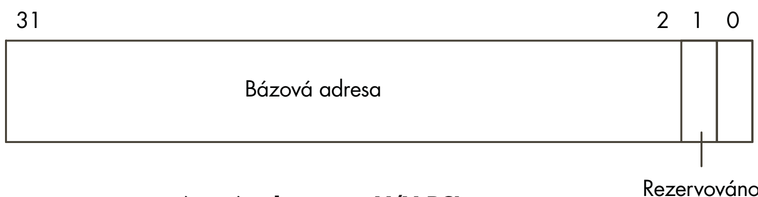
V následujícím textu je popsáno, jak příslušný kód funguje.

Zjištění kolik V/V a paměťového prostoru zařízení potřebuje

Každého nalezeného zařízení PCI se systém zeptá, kolik vyžaduje V/V prostoru a paměťového prostoru. Provede se to tak, že do bázových registrů adres se zapíší samé jedničky a poté se registry přečtou. Zařízení vrátí nuly na nevýznamných adresových bitech, čímž fakticky oznámí velikost potřebného adresového prostoru.



Bázová adresa pro paměť PCI



Bázová adresa pro V/V PCI

Obrázek 6.10
Konfigurační hlavička PCI: Bázové registry adresy

Existují dvě základní podoby bázového registru. První udává v rámci jakého adresového rozsahu musejí být umístěny registry zařízení, ať už ve V/V prostoru nebo paměťovém prostoru - to se určuje podle 0. bitu registru. Na obrázku 6.10 jsou znázorněny dvě podoby bázového registru pro paměť a V/V prostor.

Kolik adresového prostoru je zapotřebí se zjistí tak, že do bázového registru se zapíší samé jedničky a poté se jeho hodnota přečte zpět. Zařízení zapíše nuly na ty bity adresy, které je nezajímají, čímž sdělí velikost potřebného adresového prostoru. Tímto mechanismem se zajišťuje, že velikost každého adresového prostoru je vždy mocnina dvou a prostor je tak přirozeně zarovnán.

Když například inicializujete ethernetovou kartu DEC 21141, sdělí vám, že potřebuje *0x100* bajtů adresového prostoru jak ve V/V prostoru, tak v paměťovém prostoru. Inicializační kód provede alokaci požadovaného prostoru. Jakmile je prostor alokován, jsou v něm vidět řídící a stavové registry zařízení.

Přiřazení V/V a paměti můstkům a zařízením

Stejně jako veškerá paměť, i paměťový prostor PCI V/V a paměti PCI je konečný a omezený. Fixup kód na systémech jiných než Intel (a kód BIOSu na systémech Intel) musí efektivním způsobem přidělit každému zařízení jím požadovaný objem paměti. Jak V/V, tak i paměťový prostor musejí být alokovány v přirozeně zarovnaných úsecích. Pokud zařízení například požaduje V/V prostor o velikosti *0xB0*, musí být prostor zarovnán na adresu, která je násobkem *0xB0*. Kromě toho musejí být přidělované úseky V/V a paměti zarovnávány na hranice 4 KB a 1 MB. Když si k tomu přidáme, že u každého zařízení na sekundární sběrnici můstku musejí jeho adresy ležet uvnitř prostoru přidělenému primární sběrnici tohoto můstku, může být efektivní alokace prostoru obtížným úkolem.

Algoritmus používaný Linuxem počítá s tím, že každé zařízení uvedené ve stromu zařízení a sběrnic, který byl sestaven inicializačním kódem PCI, bude mít paměť přidělovánu v pořadí rostoucích adres. K průchodu datových struktur `pci_bus` a `pci_dev` se opět používá rekurzivní algoritmus. Vychází z kořene struktury PCI (ukazatel `pci_root`) a zajišťuje následující operace:

- Zarovná globální ukazatele PCI V/V prostoru a paměťového prostoru na hranice 4 KB, respektive 1 MB.
- Pro každé zařízení na aktuální sběrnici (v pořadí podle rostoucích požadavků na V/V oblast)
    - alokuje prostor ve V/V a paměti
    - o příslušný objem posune globální ukazatele V/V a paměti
    - povolí zařízení používat V/V a paměť
- Rekurzivně alokuje prostor pro všechny sběrnice pod aktuální sběrnicí. I zde dochází ke změně globálních bázových adres.

- Zarovná globální ukazatele V/V a paměti na 4 KB a 1 MB a při té příležitosti zjistí velikost a bázi V/V a paměťového okna požadovaného daným můstkem.
- Naprogramuje můstek tak, že mu oznámí bázi a velikost V/V a paměťového prostoru podřízené sběrnice.
- Aktivuje přenášení V/V a paměťových přístupů přes můstek. Znamená to, že pokud se na primární sběrnici můstku objeví požadavek na nějakou V/V nebo paměťovou adresu, která patří do okna přiřazeného sekundární sběrnici můstku, přenese můstek tento požadavek na sekundární sběrnici.

Vezměme si jako příklad PCI systém znázorněný na obrázku 6.1. Pak Fixup kód nakonfiguruje systém následujícím způsobem:

Zarovnání bází PCI Počáteční báze PCI V/V je *0x4000*, paměti *0x100000*. Díky tomu mohou můstky PCI-ISA překládat všechny adresy pod těmito hranicemi na adresové cykly sběrnice ISA.

Videozařízení Požaduje *0x200000* PCI paměti, takže tento objem naalokujeme počínaje bázovou adresou *0x200000*, protože paměťové bloky musejí být přirozeně zarovnány. Adresa paměťové báze se posouvá na *0x400000*, V/V báze zůstává na *0x4000*.

Můstek PCI-PCI Projdeme můstkem a alokujeme paměť pod ním, v této chvíli nemusíme zarovnávat báze adres, protože jsou zarovnány správně.

Ethernetové zařízení Požaduje *0xB0* bajtů V/V prostoru i paměťového prostoru. Alokujeme mu V/V prostor od báze *0x4000* a paměťový prostor od báze *0x400000*. Báze paměti se posouvá na *0x4000B0*, báze V/V se posouvá na *0x40B0*.

Zařízení SCSI Požaduje *0x1000* paměti, takže dostává (po přirozeném zarovnání) přidělenu paměť od báze *0x401000*. Báze V/V prostoru zůstává *0x40B0*, báze paměťového prostoru je *0x402000*.

Okna můstku Nyní jsme se vrátili nad můstek a nastavujeme V/V okno na interval *0x4000* až *0x40B0* a paměťové okno na *0x400000* až *0x402000*. Znamená to, že můstek bude ignorovat například přístupu k videozařízení a naopak bude předávat přístupy k SCSI nebo síťovému zařízení.

Odkazy na zdrojové texty jádra

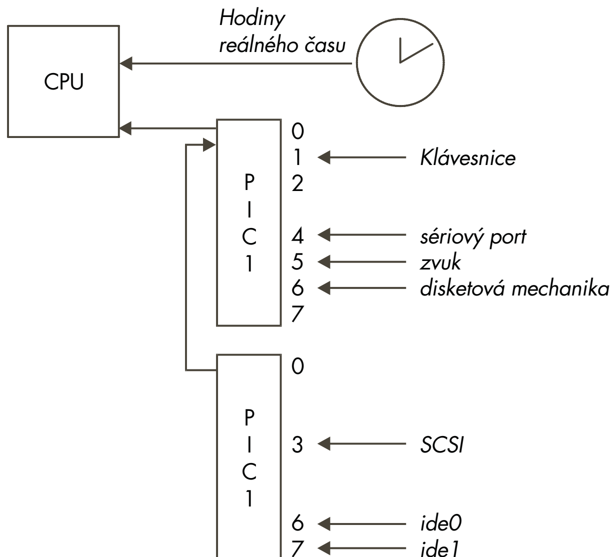
1 – `Viz include/-linux pci.h`
2 – `Viz drivers/-pci/pci.c and include/linux/-pci.h`
3 – `Viz arch/*/-kernel bios32.c`
4 – `Viz arch/*/-kernel bios32.c`
5 – `Viz Scan_bus() in drivers/pci-pci.c`
6 – `Viz arch/*/-kernel bios32.c`
7 – `Viz arch/*/-kernel/bios32.c`

Kapitola

7

Přerušení a jejich obsluha

V této kapitole se zaměříme na mechanismy, jimiž jádro Linuxu obsluhuje přerušení. I když jádro k obsluze přerušení používá obecné mechanismy a rozhraní, většina detailů v obsluze přerušení závisí na architektuře.



Obrázek 7.1
Logický diagram cesty přerušení

K vykonávání různých úkolů používá Linux různá hardwarová zařízení. Videokarta ovládá monitor, řadič IDE připojuje disk a podobně. Všechna tato zařízení je možno ovládat synchronně, což znamená, že na zařízení pošlete požadavek na nějakou operaci (řekněme na zá-

pis bloku dat z paměti na disk) a poté čekáte, až operace skončí. Tato metoda, i když by teoreticky mohla být funkční, by byla velice neefektivní a operační systém by strávil velké množství času nicneděláním, kdy by čekal na dokončení různých operací. Daleko lepší a efektivnější metoda je vznést požadavek a poté dělat něco dalšího, užitečného a později být přerušen v okamžiku, kdy zařízení požadavek splnilo. Při použití tohoto mechanismu se v systému může v jednom okamžiku provádět řada různých požadavků na různých zařízeních.

Pro přerušení činnosti procesoru musí existovat nějaká hardwarová podpora. Většina, ne-li všechny univerzální procesory, jako například Alpha AXP, používají podobnou metodu. Některé z fyzických vývodů procesoru jsou zapojeny tak, že změna napětí na těchto vývodech (řekněme z +5V na -5V) způsobí, že procesor přestane provádět to, co právě dělá, a začne provádět speciální kód pro obsluhu přerušení. Jeden z těchto vývodů může být připojen k intervalovému časovači, který bude generovat přerušení každou tisícinu sekundy, další mohou být připojeny k jiným zařízením, například k řadiči SCSI.

Většina systémů používá speciální řadič přerušení, který seskupuje dohromady přerušení od různých zařízení a předává je na jediný vývod procesoru. Tím se ušetří počet vývodů na procesoru a zároveň se zvyšuje celková flexibilita při návrhu systému. Řadič přerušení používá při své činnosti maskovací a řídicí registr. Nastavením bitů v maskovacím registru je možno zapínat a vypínat přerušení od různých zdrojů, ve stavovém registru můžeme zjistit, která přerušení jsou právě aktivní.

Některá přerušení mohou být pevně zapojena, například přerušení od časovače může být natvrdo připojeno na třetí vývod řadiče přerušení. Připojení dalších pinů může záviset na tom, jaké karty jsou zapojeny v určitých slotech ISA a PCI. Například 4. pin řadiče přerušení může být připojen k slotu PCI 0, který může jeden den sloužit k připojení síťové karty, druhý den v něm však může být zapojen řadič SCSI. Plyne z toho, že každý systém má své vlastní mechanismy předávání přerušení a operační systém musí být dostatečně pružný, aby se s tím dokázal vyrovnat.

Většina moderních univerzálních procesorů obsluhuje přerušení stejným způsobem. Když dojde k hardwarovému přerušení, CPU přeruší provádění právě aktivní instrukce a skočí na určité místo v paměti, kde je obsažen buď přímo kód obsluhy přerušení, nebo instrukce, která zajišťuje větvení obsluhy různých přerušení. Obslužný kód obvykle pracuje ve zvláštním režimu procesoru, v takzvaném *přerušovacím* režimu, a v té době nemůže za normálních okolností dojít k žádnému jinému přerušení. I zde však jsou výjimky – některé procesory například přidělují přerušením priority a povolují výskyt přerušení s vyšší prioritou. Znamená to ale, že primární kód obsluhy přerušení musí být napsán velmi pozorně a velmi často používá vlastní zásobník, který používá k uložení prováděcího stavu procesoru (tedy všech registrů a kontextu procesoru) předtím, než přejde na samotnou obsluhu události, jež přeru-

šení vyvolala. Některé procesory mají speciální skupinu registrů, které jsou přístupné pouze v přerušovacím režimu a kód obsluhy přerušení může těchto registrů využít k uložení většiny kontextu procesoru.

7.1 Programovatelné řadiče přerušení

Návrháři systémů mohou použít libovolnou architekturu přerušení podle své volby, počítače IBM PC používají obvod Intel 82C59A-2 nebo jeho deriváty - programovatelný řadič přerušení. Tento řadič se používá už od úsvitu počítačů PC a jeho registry se adresují na pevně zavedené lokace adresového prostoru sběrnice ISA. Dokonce i nejmodernější chipsety stále podporují stejné typy registrů na stejných místech v paměti. Systémy založené na jiném procesoru než Intel, například Alpha AXP, nejsou těmito architektonickými omezeními svázány, a tak často používají jiné řadiče přerušení.

Na obrázku 7.1 vidíme dva 8bitové řadiče spřažené dohromady, každý má vlastní maskovací a stavový registr, PIC1 a PIC2. Maskovací registry se mapují na adresy *0x21* a *0xA1*, stavové registry na adresy *0x20* a *0xA0*. Zapsáním jedničky na určitý bit maskovacího registru se přerušení povoluje, zapsáním nuly se vypíná. Tedy zápis jedničky na bit 3 povolí přerušení 3, nula na stejném bitu toto přerušení deaktivuje. Je bohužel nepříjemné, že maskovací registry jsou navrženy pouze pro zápis, nemůžete z nich zpět načíst hodnotu, která je v nich zapsána. Znamená to, že Linux si musí vést lokální kopii nastavení maskovacích registrů. V rutinách pro aktivaci a deaktivaci přerušení modifikuje tyto uložené hodnoty a při každém zápisu do registru zapisuje celou hodnotu masky.

Když dojde k výskytu přerušení, přečte obslužný kód přerušení dva stavové registry přerušení (ISR). Registr na adrese *0x20* je chápán jako nižších osm bitů 16bitového přerušovacího registru, registr na adrese *0xA0* jako vyšších osm bitů. Takže přerušení na prvním bitu registru na adrese *0xA0* bude chápáno jako přerušení 9. Bit 2 řadiče PIC1 není k dispozici, protože slouží k řetězení přerušení od řadiče PIC2. Každé přerušení na řadiči PIC2 se projeví jako přerušení na druhém bitu řadiče PIC1.

7.2 Inicializace datových struktur obsluhy přerušení

Datové struktury jádra pro obsluhu přerušení inicializují ovladače zařízení podle svých požadavků na řízení systémových přerušení. K tomu účelu používají ovladače skupinu služeb jádra, které slouží k žádosti o přerušení, k jejich aktivaci a deaktivaci.

Jednotlivé ovladače zařízení prostřednictvím těchto rutin registrují adresy svých obslužných rutin přerušení.

Některá přerušení jsou pevně dána konvencemi architektury PC, takže ovladače při své inicializaci prostě jen požádají o toto přerušení. To se týká například ovladače disketových mechanik, které vždy používají přerušení IRQ 6. Za jiných okolností nemusí ovladač zařízení vědět, jaké přerušení jeho zařízení používá. Tento problém se neobjevuje u ovladačů zařízení PCI, které vždy znají číslo přerušení používané jejich zařízením. Bohužel však neexistuje jednoduchý způsob, jak mohou číslo přerušení svého zařízení zjistit ovladače zařízení ISA. Linux tento problém řeší tak, že umožňuje ovladačům vyhledat číslo přerušení svého zařízení.

Nejprve ovladač provede nějakou akci, která vyvolá přerušení od zařízení. Poté se povolí všechna nepřiřazená přerušení. Znamená to, že přerušení od našeho zařízení nyní bude prostřednictvím programovatelného řadiče přerušení doručeno. Linux přečte obsah stavového registru přerušení a předá jej ovladači zařízení. Nenulový výsledek znamená, že v průběhu testu se objevilo jedno nebo více přerušení. Ovladač zruší režim hledání a nepřiřazená přerušení se opět zakáží.

Pokud se ovladači zařízení ISA podařilo nalézt číslo přerušení jeho zařízení, může nyní normálním postupem požádat o řízení tohoto přerušení.

Systémy PCI jsou podstatně dynamičtější než systémy ISA. Číslo přerušení, které bude zařízení ISA používat, se velmi často nastavuje jumpery přímo na řadiči a ovladač zařízení musí toto číslo znát. Naproti tomu zařízení PCI dostávají čísla přerušení přidělena PCI BIOSem nebo subsystémem PCI při inicializaci sběrnice PCI při zavádění systému. Každé zařízení PCI může použít jednu ze čtyř pozic přerušení, A, B, C nebo D. Toto nastavení je dáno výrobcem zařízení a většina zařízení PCI implicitně používá přerušovací pozici A. Pak je možno směrovat pozici A slotu PCI 4 na šestý pin řadiče přerušení, pozici B slotu 4 na sedmý pin řadiče a tak dále.

Směrování přerušení PCI je dáno výhradně architekturou systému, a proto musí být k dispozici nějaký kód, který rozumí topologii směrování přerušení PCI. Na systémech Intel to zajišťuje kód BIOSu, který se provádí při zavádění systému, na systémech bez BIOSu (například na platformě Alpha AXP) však tuto funkci plní jádro Linuxu.

Inicializační kód PCI zapisuje číslo pinu řadiče přerušení do konfigurační hlavičky každého zařízení PCI. Čísla přerušení stanovuje na základě znalosti směrovací topologie přerušení PCI, na znalosti čísel slotů PCI a jimi používaných přerušení PCI. Číslo přerušení používané zařízením je pak pevně dáno a je uloženo v konfigurační hlavičce tohoto zařízení. Tyto informace se zapisují do položky *interrupt line*. Když se spustí ovladač zařízení, přečte si tuto informaci a může jádro Linuxu požádat o předání kontroly nad daným přerušením.

V systému se může nacházet více zdrojů přerušení PCI, například pokud se používají můstky PCI-PCI. Počet zdrojů přerušení může přesáhnout počet dostupných pinů programovatelného řadiče přerušení. V takovém případě mohou zařízení PCI sdílet přerušení, tedy jeden pin řadiče přerušení přijímá přerušení od více zařízení. Tento mechanismus Linux podporuje tak, že první zařízení žádající o přidělení přerušovacího pinu oznamuje, zda se jeho pin může sdílet. Sdílení přerušení vede k více datovým strukturám `irqaction`, na něž všechny ukazuje jeden vektor `irq_action`. Když se objeví sdílené přerušení, Linux bude volat všechny obslužné kódy tohoto přerušení. Každý ovladač zařízení, který může sdílet přerušení (což by měly být všechny ovladače zařízení PCI) musí být připraven na to, že se bude volat obslužná rutina přerušení i v případě, že není co obsluhovat.

7.3 Obsluha přerušení

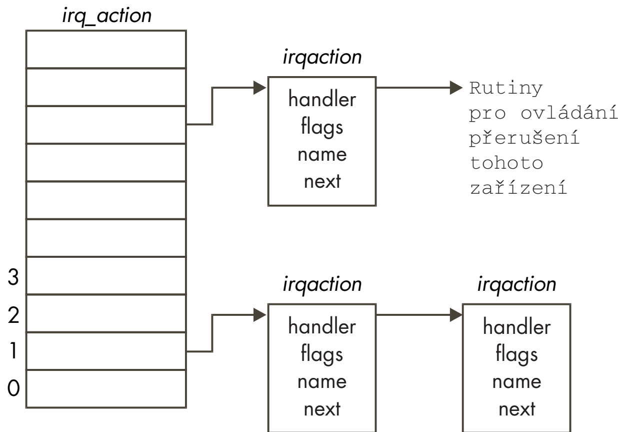
Jedním ze základních úkolů subsystému obsluhy přerušení v Linuxu je nasměrování přerušení na správnou obslužnou rutinu. Tento kód musí rozumět topologii přerušení v systému. Pokud například řadič disketové mechaniky používá šestý pin řadiče přerušení,[1] pak řadič přerušení musí toto přerušení detekovat jako přerušení od disketového řadiče a musí je předat kódu obsluhy přerušení od diskety. Linux používá sadu ukazatelů na datové struktury obsahující adresy obslužných rutin přerušení. Tyto rutiny jsou součástí ovladačů zařízení připojených k systému a každý ovladač musí v době své inicializace požádat o obsluhu svého přerušení. Na obrázku 7.2 je vidět strukturu `irq_action`, která je vektorem ukazatelů na struktury `irqaction`. Každá datová struktura `irqaction` obsahuje informace o obsluze svého přerušení včetně adres obslužné rutiny tohoto přerušení. Protože se počty přerušení a jejich obsluha mohou na různých architekturách nebo i na různých systémech se stejnou architekturou lišit, je obslužný kód přerušení Linuxu závislý na architektuře. Znamená to, že velikost vektoru `irq_action` se liší podle počtu možných zdrojů přerušení.

Když se objeví přerušení, Linux musí nejprve zjistit jeho zdroj tím, že přečte stavový registr programovatelného řadiče přerušení systému. Zdroj přerušení pak přeloží na offset ve vektoru `irq_action`. Takže například přerušení na šestém pinu řadiče přerušení (od ovladače disket) bude přeloženo na sedmý ukazatel ve vektoru obsluhy přerušení. Pokud pro dané přerušení není zaveden žádný obslužný kód, ohlásí jádro Linuxu chybu, v opačném případě postupně zavolá všechny obslužné kódy daného přerušení.

Když jádro Linuxu zavolá obslužnou rutinu přerušení nějakého ovladače zařízení, musí rutina rychle zjistit proč k přerušení došlo a vhodně na to zareagovat. K zjištění příčiny přerušení čte ovladač zařízení stavový registr svého zařízení. Zařízení může hlásit chybu nebo může oznamovat, že operace skončila. Například řadič disketové mechaniky může hlásit, že čtecí

[1] Zrovna řadič disketových mechanik v systémech PC používá fixně přidělené přerušení a podle zavedené konvence je vždy připojen na přerušení 6.

hlavička je vystavena nad požadovaným sektorem diskety. Jakmile je zjištěn důvod přerušení, může ovladač zařízení potřebovat provést další obsluhu. Pokud to je nutné, má jádro Linuxu mechanismy, které umožňují odložit tuto práci na později. Tím se zabrání, aby procesor strávil příliš mnoho času v přerušovacím režimu. Další podrobnosti naleznete v kapitole o ovladačích zařízení.



Obrázek 7.2
Datové struktury obsluhy přerušení

Odkazy na zdrojové texty jádra

1 – Viz request_irq(), enable_irq() and disable_irq() in arch/*/kernel irq.c

2 – Viz irq_probe_*() in arch/*/kernel/-irq.c

3 – Viz arch/alpha/-kernel/biow32.c

Kapitola

8

Ovladače zařízení

Jedním z úkolů operačního systému je izolovat uživatele od specifik hardwarových zařízení. Například virtuální souborový systém nabízí uniformní pohled na všechny připojené souborové systémy bez ohledu na příslušná fyzická zařízení. V této kapitole popisujeme, jak Linux spravuje fyzická zařízení v systému.

Procesor není jediným inteligentním zařízením v systému, každé fyzické zařízení má svůj vlastní hardwarový řadič. Klávesnice, myš a sériové porty jsou řízeny vstupně/výstupním čipem, disky IDE řadičem IDE, disky SCSI řadičem SCSI a tak dále. Každý hardwarový řadič má své řídící a stavové registry, které se pro různá zařízení liší. Řídící registry řadiče SCSI Adaptec 2940 jsou úplně jiné než registry řadiče SCSI NCR 810. Řídící registry slouží ke spouštění a zastavení zařízení, k jeho inicializaci a diagnostice problémů. Namísto toho, aby obslužný kód jednotlivých zařízení v systému obsahovala každá aplikace, je tento kód přítomen pouze v jádře systému. Program, který obsluhuje hardwarový řadič, se označuje jako ovladač zařízení. Ovladače zařízení v Linuxu jsou v zásadě sdílené knihovny privilegovaných, paměťově rezidentních nízkoúrovňových rutin pro obsluhu hardwaru. Právě ovladače zařízení skrývají odlišnosti mezi různými zařízeními.

Jedna ze základních funkcí je abstrakce obsluhy zařízení. Všechna hardwarová zařízení vypadají jako normální soubory, dají se otevírat, zavírat, číst a zapisovat pomocí stejných standardních systémových volání, jaká se používají pro soubory. Každé zařízení v systému je reprezentováno *souborem zařízení*, například první disk IDE je reprezentován souborem `/dev/hda`. Pro bloková zařízení (disky) a znaková zařízení se tyto soubory vytvářejí příkazem `mknod` a popisují zařízení pomocí hlavního a vedlejšího čísla zařízení. Síťová zařízení jsou rovněž reprezentována soubory zařízení, ty však vytváří přímo Linux, když síťové zařízení nalezne. Všechna zařízení ovládaná společným ovladačem zařízení mají stejné hlavní číslo. Vedlejší čísla pak slouží k odlišení těchto zařízení a jejich řadičů, například každá oblast

primárního disku IDE má jiné vedlejší číslo zařízení. Takže například /dev/hda2, druhá oblast primárního disku IDE, má hlavní číslo 3 a vedlejší číslo 2. Linux mapuje soubor zařízení předávaný v systémovém volání (například při připojování souborového systému na blokovém zařízení) na ovladač zařízení pomocí hlavního čísla zařízení a řady systémových tabulek, například tabulky znakových zařízení, chrdevs.

Linux podporuje tři typy hardwarových zařízení: znakové, blokové a síťové. Znaková zařízení se čtou a zapisují přímo bez použití bufferů, jsou to například sériové porty /dev/cua0 a /dev/cua1. Bloková zařízení je možno číst a zapisovat pouze v násobcích velikosti bloku, který typicky bývá 512 nebo 1 024 bajtů. K blokovým zařízením se přistupuje přes buffery a je možno k nim přistupovat náhodně, tedy je možno přečíst nebo zapsat libovolný blok bez ohledu na jeho polohu na zařízení. K blokovým zařízením je možno přistupovat přes příslušný soubor zařízení, daleko častěji se k nim však přistupuje přes souborový systém. Pouze bloková zařízení podporují připojení souborových systémů. K síťovým zařízením se přistupuje přes soketové rozhraní BSD a síťový subsystém popsaný v kapitole „Sítě“.

V jádře Linuxu existuje řada různých ovladačů zařízení (mimo jiné i v tom je síla Linuxu), všechny však mají určité společné rysy:

kód jádra Ovladače zařízení jsou částí jádra a stejně jako ostatní kód v jádře, pokud by fungovaly chybně, mohly by vážně poškodit celý systém. Špatně napsaný ovladač zařízení může zhroutit celý systém, poškodit souborový systém a zničit data.

rozhraní jádra Ovladače zařízení musejí jádru Linuxu nebo subsystému, jehož jsou součástí, poskytovat standardní rozhraní. Například terminálové zařízení poskytuje jádru souborové vstupně/výstupní rozhraní, zařízení SCSI poskytuje rozhraní zařízení SCSI pro subsystém SCSI, který pak jádru dále poskytuje jednak souborové vstupně/výstupní rozhraní a jednak bufferové rozhraní.

mechanismy a služby jádra Ovladače zařízení používají ke své činnosti standardní služby jádra, jako je alokace paměti, doručování přerušení a čekací fronty.

modularita Většina ovladačů zařízení v Linuxu může být nahrána podle potřeby, když je nějaký modul jádra potřebuje, a odstraněna, když nejsou více zapotřebí. Díky tomu je jádro velmi přizpůsobivé a efektivně využívá systémových prostředků.

konfigurovatelnost Ovladače zařízení mohou být vestavěny přímo do jádra. Které z ovladačů vestavět, se konfiguruje při překladu jádra.

dynamičnost Když se systém zavádí a inicializují se jednotlivé ovladače zařízení, každý ovladač hledá hardwarové zařízení, které má ovládat. Nevadí, pokud zařízení ovládané nějakým určitým ovladačem neexistuje. V takovém případě je ovladač prostě nadbytečný a nezpůsobí žádnou škodu kromě toho, že zabírá část systémové paměti.

8.1 Dotazování a přerušení

Vždy, když ovladač zařízení vydá nějaký povel, například „*přesuň hlavičku na 42. stopu diskety*“, může si ovladač zvolit metodu jak zjistí, že operace byla dokončena. Ovladač se může buď zařízení dotazovat, nebo může použít přerušení.

Dotazování typicky znamená, že ovladač opakovaně čte řídicí registr zařízení tak dlouho, až se status zařízení změní a indikuje dokončení operace. Protože ovladač zařízení je součástí jádra, bylo by nevhodné, kdyby se delší dobu dotazoval, protože nic jiného v jádře by nemohlo pracovat až do doby, než by byl požadavek splněn. Namísto cyklického dotazování používají ovladače zařízení časovač, který způsobí, že jádro periodicky volá nějakou rutinu ovladače. Obsluha časovače bude tak periodicky testovat status zařízení. Tento mechanismus používá Linux pro obsluhu disketových mechanik. Dotazování pomocí časovačů je efektivnější řešení, nejefektivnější je však použití přerušení.

Přerušením řízený ovladač zařízení se používá pro zařízení, která v době, kdy potřebují nějakou obsluhu, vyvolají přerušení. Například ovladač ethernetové karty vyvolá přerušení, když karta obdrží paket ze sítě. Jádro Linuxu musí být schopno doručit přerušení od hardwarového zařízení správnému ovladači zařízení. Dosahuje se toho tím, že ovladače zařízení v jádře registrují používaná přerušení. Registruje se adresa obslužné rutiny přerušení a číslo přerušení, které chce ovladač obsluhovat. V souboru `/proc/interrupts` můžete zjistit počet přerušení v systému a jejich obsluhu ovladači zařízení:

```
 0:      727432   timer
 1:       20534   keyboard
 2:           0   cascade
 3:       79691 + serial
 4:       28258 + serial
 5:           1   sound blaster
11:       20868 + aic7xxx
13:           1   math error
14:         247 + ide0
15:         170 + ide1
```

Žádosti o přerušení ovladač registruje v době své inicializace. Některá přerušení v systému jsou pevně dána konvencemi architektury IBM PC. Například ovladače disket vždy používají přerušení 6. Jiná přerušení, například přerušení od zařízení PCI, se přidělují dynamicky při startu systému. V takovém případě musí ovladač nejprve zjistit číslo přerušení (IRQ) svého zařízení a teprve potom může žádat o jeho obsluhu. Pro přerušení Linux podporuje standardní volání PCI BIOSu, která umožňují zjistit informace o zařízeních v systému včetně jim přidělených čísel přerušení.

Mechanismus doručení přerušení procesoru je závislý na architektuře, nicméně na většině architektur se přerušení doručují ve speciálním režimu, který zabrání výskytu jiných přerušení v systému. Ovladač zařízení by měl při obsluze přerušení pracovat co nejrychleji, aby jádro mohlo záhy prohlásit přerušení za obsloužené a mohlo se vrátit k přerušené práci. Ovladače zařízení, které při výskytu přerušení potřebují provést větší objem práce, mohou použít obslužný mechanismus bottom-half nebo frontu úloh, čímž zajistí, aby se potřebná obsluha volala později v normálním režimu.

8.2 Přímý přístup do paměti (DMA)

Použití přerušením řízených ovladačů zařízení funguje dobře, pokud se přenášejí rozumně nízké objemy dat. Například modem o rychlosti 9 600 Baudů přenese přibližně jeden znak za jednu milisekundu. Pokud je latentní doba přerušení, tedy čas mezi tím než zařízení vyvolá přerušení a než se předá řízení přerušovací obsluze, malá (řekněme 2 milisekundy), je celkový dopad datového přenosu na výkon systému velmi nízký. Přenos dat prostřednictvím modemu o rychlosti 9 600 Baudů spotřebuje pouhých 0,002% procesorového času. U vysokorychlostních zařízení, jako jsou například pevné disky nebo síťové karty, je zatížení přenosem dat podstatně vyšší. Rozhraní SCSI je schopno za sekundu přenést až 40 MB dat.

K vyřešení tohoto problému byl vyvinut mechanismus přímého přístupu do paměti, DMA. řadič DMA umožňuje zařízením přenášet data do nebo ze systémové paměti bez zásahu procesoru. Řadič ISA DMA na počítačích PC má 8 kanálů DMA, z nichž 7 je přístupných pro potřeby ovladačů zařízení. Každému kanálu DMA je přiřazen 16bitový adresový registr a 16bitové počitadlo. K zahájení datového přenosu musí ovladač zařízení nejprve nastavit adresový registr a čítač a dále musí určit směr přenosu, zápis či čtení. Pak oznámí zařízení, že jakmile bude chtít, může zahájit přenos DMA. Po skončení přenosu vyvolá zařízení přerušení. Zatímco přenos probíhá, procesor není zatěžován a může se věnovat jiné činnosti.

Ovladače zařízení musejí s DMA spolupracovat velmi opatrně. Řadič DMA v první řadě neví nic o virtuální paměti, má přístup přímo k fyzické paměti systému. Data přenášená pomocí DMA tedy musí být uložena jako souvislý blok fyzické paměti. Znamená to, že DMA nemůžete použít přímo ve virtuálním adresovém prostoru procesu. Můžete nicméně zamknout fyzic-

ké stránky procesu v paměti, takže po dobu přenosu DMA nemůže dojít k jejich odložení. Dále řadič DMA nemůže přistupovat k celé fyzické paměti. Adresový registr řadiče DMA reprezentuje prvních 16 bitů adresy přenosu, dalších 8 bitů je dáno stránkovým registrem. Znamená to, že přenosy DMA mohou probíhat pouze v prvních 16 MB fyzické paměti.

Kanály DMA jsou vzácná zařízení, protože je jich pouze sedm a nedají se mezi zařízeními sdílet. Stejně jako s přerušeními, musí být ovladač zařízení schopen určit, který kanál DMA může použít. Obdobně jako u přerušení, některá zařízení mají kanál DMA pevně dán. Například disketový řadič používá vždy kanál DMA 2. Někdy je možno kanály DMA zařízení nastavovat pomocí přepínačů, tento způsob používá řada síťových karet. Modernějším zařízením je možno (prostřednictvím jejich řídicích registrů) říci, jaký kanál DMA mají použít a v takovém případě si ovladač zařízení prostě zvolí jeden z volných kanálů DMA.

Linux sleduje využití kanálů DMA pomocí vektoru datových struktur `dma_chan` (jedna pro každý kanál DMA). Datová struktura `dma_chan` obsahuje pouze dvě položky, ukazatel na řetězec popisující vlastníka kanálu DMA a příznak, zda kanál DMA je či není přidělen. Když vypisujete soubor `/proc/dma`, vypisuje se obsah právě tohoto vektoru.

8.3 Paměť

Ovladače zařízení musejí být při práci s pamětí velmi opatrné. Protože jsou součástí jádra, nemohou používat virtuální paměť. Vždy, když je ovladač zařízení spuštěn, ať už po přijetí přerušení nebo mechanismem bottom-half nebo obsluhou fronty úloh, může být aktuální proces jiný. Ovladač zařízení se nemůže spoléhat na to, že zrovna poběží určitý proces, i když pracuje jeho jménem. Stejně jako zbytek jádra, i ovladače zařízení používají různé datové struktury k uložení stavu zařízení, které ovládají. Tyto datové struktury mohou být alokovány staticky jako součást ovladače, to by však bylo plýtvání, protože jádro by bylo větší, než je nezbytně nutné. Většina ovladačů zařízení si pro uložení svých dat alokuje nestránkovanou paměť jádra.

Linux poskytuje rutiny pro alokaci a dealokaci paměti jádra a tyto rutiny využívají právě ovladače zařízení. Paměť jádra se alokuje v úsecích, jejichž velikost je vždy mocninou dvou. Například 128 nebo 512 bajtů, dokonce i v případě, že ovladač zařízení požaduje méně. Velikost paměti požadovaná ovladačem se zaokrouhlí nahoru na nejbližší hranici. Tím se usnadňuje dealokace paměti jádra, protože menší volné bloky je možno snáze spojovat do větších.

Může se stát, že při požadavku na paměť jádra bude zapotřebí provést větší množství práce. Pokud je málo volné paměti, může být nutné zrušit nebo odložit nějaké fyzické stránky. Za normálních okolností Linux žadatele pozastaví a umístí jej do čekací fronty, dokud nebude dostatek paměti. Ne všechny ovladače paměti (nebo přímo samotný kód jádra) si to vždy mohou dovolit, takže alokační rutiny paměti jádra je možno volat také tak, aby v případě, že nebudou schopny paměť poskytnout okamžitě, vyvolaly chybu. Pokud ovladač zařízení žádá

o paměť pro potřeby přenosu DMA, může specifikovat, že paměť bude využita pro DMA. Díky tomu potřebuje znát podrobnosti o přidělování paměti pro potřeby DMA pouze jádro a ne samotný ovladač.

8.4 Komunikace ovladačů s jádrem

Jádro Linuxu musí být schopno komunikovat s ovladači standardními způsoby. Každá třída ovladačů, znakových, blokových a síťových, poskytuje jednotné rozhraní, které jádro používá při požadavcích na služby. Toto společné rozhraní zajišťuje, že jádro může ve většině případů komunikovat s ovladači zcela rozdílných zařízení úplně stejně. Například disky SCSI a IDE se chovají velmi rozdílně, jádro Linuxu však při komunikaci s oběma z nich používá stejné rozhraní.

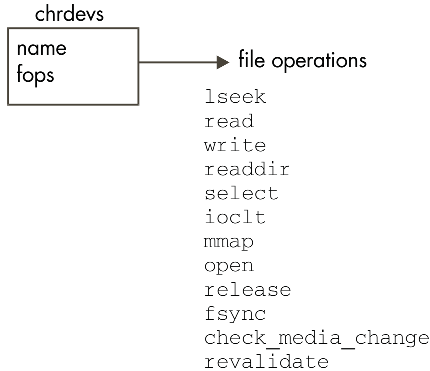
Linux je značně dynamický, vždy když se jádro Linuxu zavádí, může detekovat různá fyzická zařízení a může tedy potřebovat různé ovladače zařízení. Linux umožňuje zahrnout ovladače zařízení přímo do jádra prostřednictvím konfiguračních skriptů jádra. Když se takové ovladače při zavádění inicializují, mohou zjistit, že nemají žádné zařízení, které budou ovládat. Další ovladače je možno nahrávat do jádra podle potřeby. Aby se dalo s touto dynamickou podstatou ovladačů zařízení pracovat, musejí se ovladače vždy při inicializaci zaregistrovat. Linux udržuje tabulky registrovaných ovladačů zařízení jako součást rozhraní pro komunikaci s nimi. Tyto tabulky obsahují ukazatele na rutiny a informace, které slouží pro podporu rozhraní jednotlivých tříd ovladačů.

8.4.1 Znaková zařízení

Ke znakovým zařízením, nejjednodušším zařízením Linuxu, se přistupuje stejně jako k souborům - aplikace k jejich otevření, čtení a zápisu používají stejná standardní systémová volání jako kdyby šlo o soubory. Platí to i v případech, kdy se jedná například o modem, který prostřednictvím démona PPP slouží k připojení systému na síť. Když se znakové zařízení inicializuje, jeho ovladač se v jádře Linuxu registruje přidáním položky do vektoru `chrdevs` datových struktur `device_struct`. Hlavní identifikátor zařízení (například 4 pro zařízení `tty`) slouží jako index tohoto vektoru. Hlavní identifikátor je pro zařízení pevně dán.

2

Každá položka vektoru `chrdevs`, datová struktura `device_struct`, obsahuje dva prvky: ukazatel na jméno registrovaného ovladače zařízení a ukazatel na blok souborových operací. Blok souborových operací představuje adresy rutin v ovladači zařízení, které obsluhují jednotlivé souborové operace jako otevření, čtení, zápis a zavření. Obsah souboru `/proc/devices` se pro znaková zařízení pořizuje právě z vektoru `chrdevs`.



Obrázek 8.1
Znakové zařízení

Když dojde k otevření speciálního znakového souboru reprezentujícího znakové zařízení (například souboru `/dev/cua0`), musí jádro vše zařídit tak, aby se volaly souborové rutiny správného ovladače znakového zařízení. Stejně jako u normálních souborů nebo adresářů, každý soubor zařízení má rovněž svůj VFS inode. Inode pro soubor znakového zařízení, respektive pro soubor každého zařízení, obsahuje jak hlavní, tak vedlejší identifikátor zařízení. VFS inode byl vytvořen nějakým souborovým systémem na nižší úrovni, například systémem `ext2`, z informací v reálném souborovém systému, když byl speciální soubor zařízení nalezen.

3

Každý VFS inode má přiřazenu množinu souborových operací, které se liší podle toho, jaký objekt souborového systému příslušný inode reprezentuje. Když se vytváří VFS inode reprezentující soubor znakového zařízení, nastaví se souborové operace na implicitní operace znakového zařízení.

Z implicitních operací je definována pouze jedna - operace otevření. Když aplikace otevře soubor znakového zařízení, obecná obslužná rutina otevření v jádře použije hlavní identifikátor zařízení jako index do vektoru `chrdevs` a získá blok souborových operací pro toto zařízení. Dále vytvoří datovou strukturu `file`, která popisuje soubor znakového zařízení a její ukazatel souborových operací nasměruje na ovladač zařízení. Následně se všechny souborové operace v aplikaci budou mapovat na volání rutin v ovladači zařízení.

4

5

8.4.2 Bloková zařízení

Bloková zařízení rovněž podporují stejné metody přístupu jako k souborům. Mechanismus používaný k přiřazení správné skupiny souborových operací otevřenému souboru zařízení funguje velmi podobně jako u znakových zařízení. Registrovaná bloková zařízení Linux ukládá ve vektoru `blkdevs`. Tento vektor, stejně jako vektor `chrdevs`, je indexován pomocí hlavního čísla zařízení. Jeho položkami jsou rovněž datové struktury `device_struct`. Na

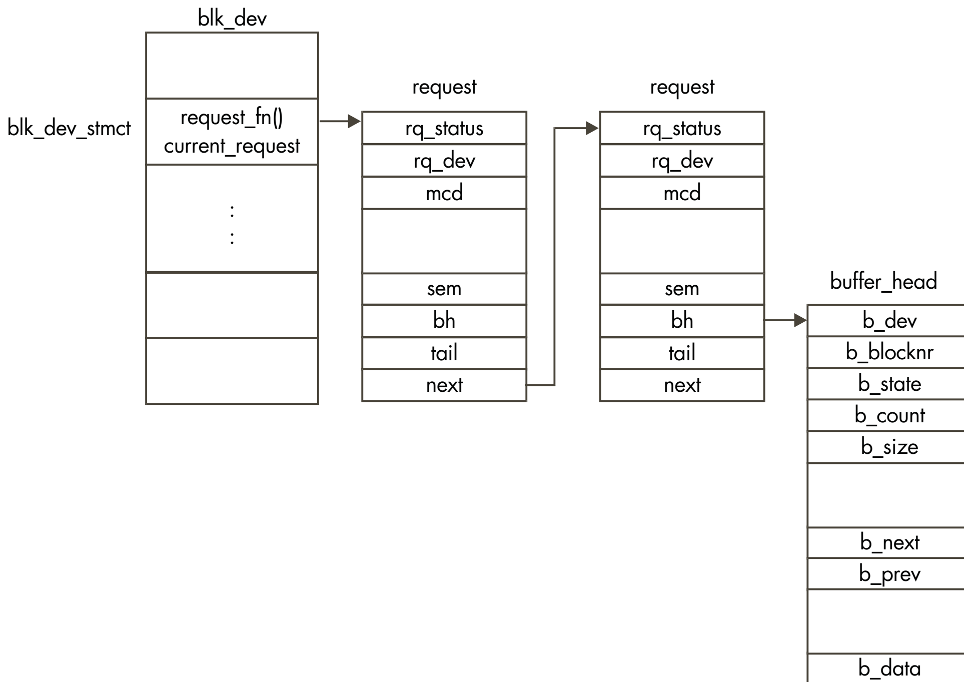
6

rozdíl od znakových zařízení jsou bloková zařízení rozdělena do tříd. Například zařízení SCSI patří do jedné třídy, zařízení IDE do jiné. V jádře Linuxu se registrují právě třídy, které pak zajišťují souborové operace. Ovladač zařízení jedné třídy blokových zařízení poskytuje rozhraní specifické pro danou třídu. Takže například ovladač konkrétního zařízení SCSI poskytuje rozhraní subsystému SCSI, které pak subsystém SCSI používá k poskytnutí rozhraní souborových operací jádru.

Každé blokové zařízení musí poskytovat rozhraní k bufferovým operacím a také normální rozhraní souborových operací. Každý ovladač blokového zařízení vyplňuje své údaje v datové struktuře `blk_dev_struct` vektoru `blk_dev`. Tento vektor se opět indexuje hlavním číslem zařízení. Datová struktura `blk_dev_struct` se skládá z adresy rutiny žádostí a ze seznamu datových struktur `request`, z nichž každá reprezentuje jednu žádost o čtení nebo zapsání bloku dat na zařízení.

7

8;



Obrázek 8.2
Žádosti bufferů blokových zařízení

Vždy, když vyrovnávací paměť bufferů chce přečíst nebo zapsat blok dat z nebo na registrované zařízení, přidává do příslušné struktury `blk_dev_struct` další strukturu `request`. Na obrázku 8.2 vidíme, že každá žádost má ukazatele na jeden nebo více datových struktur `buffer_head`, z nichž každá znamená jednu žádost o čtení nebo zápis blo-

ku dat. Datové struktury buffer_head jsou uzamčeny (vyrovnávací pamětí bufferů) a může existovat proces, čekající na dokončení blokové operace v tomto bufferu. Každá struktura request se alokuje ze statického seznamu all_requests. Když se přidává žádost do prázdného seznamu žádostí, volá se funkce ovladače, která zahajuje zpracování fronty žádostí. Ovladač pak jednoduše zpracuje všechny žádosti ve frontě.

Když ovladač zařízení dokončí zpracování žádosti, musí odstranit všechny struktury buffer_head ze struktury request, označí je jako aktuální a odemkne je. Tímto odemknutím struktur buffer_head dojde k probuzení procesu, který čekal na dokončení blokové operace. Příkladem může být situace, kdy se hledá jméno souboru a souborový systém ext2 musí z blokového zařízení, na němž je souborový systém uložen, načíst blok dat, který obsahuje další adresářovou položku souborového systému. Proces spí na datové struktuře buffer_head až do doby, než jej ovladač zařízení probudí. Datová struktura request se označí jako volná, takže ji bude možno použít pro uložení dalšího požadavku na blokové zařízení.

8.5 Pevné disky

Pevné disky představují trvalou metodu uložení dat, která ukládají na rotující diskové povrchy. Při zápisu dat malinká hlavička zmagnetizuje nepatrnou část magnetického povrchu. Při čtení dat hlavička detekuje, zda je příslušný kousíček povrchu zmagnetován nebo ne.

Disková jednotka se skládá z jednoho nebo více kotoučů, které jsou vyrobeny z leštěného skleněného nebo keramického materiálu a jsou pokryty tenkou vrstvičkou oxidu železa. Jednotlivé kotouče jsou připevněny ke společné ose a rotují konstantní rychlostí, která může být od 3 000 do 10 000 ot./min. podle typu disku. Srovnejte tuto rychlost s disketou, která se otáčí rychlostí 360 ot./min. Čtecí/zápisová hlavička zodpovídá za čtení a zápis dat na povrch kotouče, pro každý kotouč jsou dvě hlavičky, každá pracuje na jednom povrchu. Hlavičky se fyzicky nedotýkají povrchu kotouče, vznášejí se na vzduchovém polštáři ve vzdálenosti kolem několika nanometrů. Vystavovací rameno zajišťuje pohyb hlaviček nad povrchem kotouče. Všechny hlavičky jsou vystavovány společně, pohybují se nad kotouči vždy stejně.

Každý povrch je rozdělen na soustředné kruhové prstence zvané *stopy*. Stopa 0 je nejblíže vnějšímu okraji povrchu, stopa s nejvyšším číslem je nejblíže ke středu povrchu. *Cylindr* je skupina stop se stejným číslem. Takže například všechny páté stopy na obou stranách všech kotoučů se označují jako pátý cylindr. Protože počet cylindrů je stejný jako počet stop, geometrie disku se často popisuje v cylindrech. Každá stopa je rozdělena na *sektory*. Sektor je nejmenší datová jednotka, kterou je možno z disku najednou přečíst nebo zapsat a odpovídá tak velikosti bloku. Sektory jsou obvykle velké 512 bajtů a jejich velikost zpravidla nastavuje při fyzickém formátování disku jeho výrobce.

Disky se většinou popisují svou geometrií; počtem cylindrů, hlaviček a sektorů. Například v době zavádění může Linux popisovat jeden z disků IDE takto:

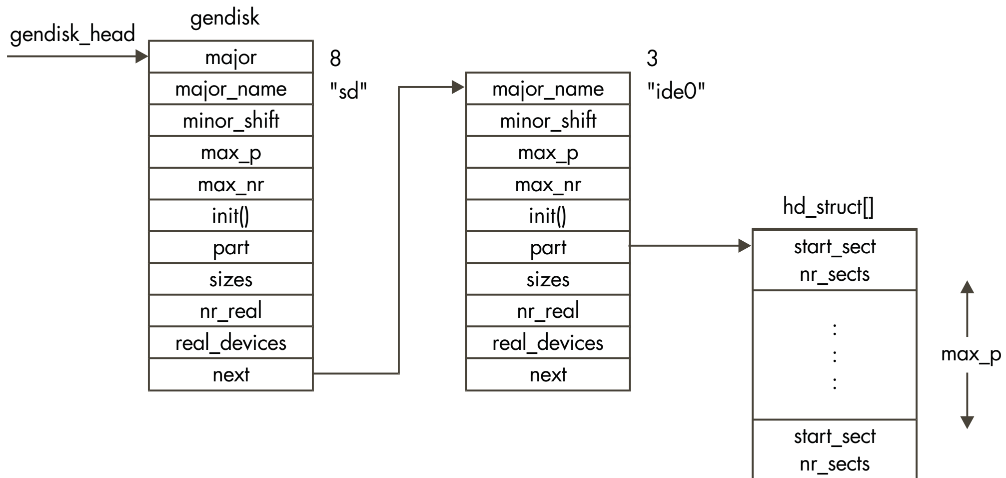
```
hdb: Conner Peripherals 540MB - CFS540A, 516MB w/64kB Cache,
  CHS=1050/16/63
```

Znamená to, že disk má 1 050 cylindrů (stop), 16 hlaviček (8 kotoučů) a 63 sektorů na každé stopě. Při velikosti sektoru 512 bajtů nám to dává celkovou kapacitu disku 529 200 bajtů. To ovšem neodpovídá hlášené velikosti 516 MB, protože některé sektory se používají k uložení rozdělovacích informací. Některé disky automaticky nalézají vadné sektory a reindexují disk tak, aby se vyhnul jejich použití.

Pevné disky se dále dělí na *oblasti*. Oblast je velká oblast sektorů, přidělená pro určitý účel. Rozdělením operačního disku na oblasti je možno tento disk používat několika operačními systémy pro různé účely. Řada linuxových systémů používá jeden disk se třemi oblastmi: na jedné je souborový systém DOS, na druhé systém ext2 a třetí slouží jako odkládací oblasti. Rozdělení disku na oblasti je popsáno rozdělovací tabulkou (partition table), každá položka této tabulky popisuje začátek a konec jedné oblasti v číslech hlaviček, sektorů a cylindrů. U disků formátovaných systémem DOS, tedy rozdělených příkazem `fdisk`, existují čtyři primární diskové oblasti. Ne všechny čtyři položky rozdělovací tabulky musejí být použity. `fdisk` podporuje tři typy partic: primární, rozšířené a logické. Rozšířené oblasti nejsou oblastmi v pravém slova smyslu, obsahují pouze několik logických oblastí. Rozšířené a logické oblasti se používají jako metoda obejití omezení pouhých čtyř primárních oblastí. Následující výpis příkazu `fdisk` pochází z disku, který je rozdělen na dvě oblasti:

```
Disk /dev/sda: 64 heads, 32 sectors, 510 cylinders
Units = cylinders of 2048 * 512 bytes

   Device Boot    Begin    Start    End   Blocks   Id   System
/dev/sda1             1        1    478   489456   83   Linux native
/dev/sda2           479      479    510    32768   82   Linux swap

Expert command (m for help): p

Disk /dev/sda: 64 heads, 32 sectors, 510 cylinders

Nr   AF  Hd  Sec  Cyl   Hd   Sec   Cyl   Start    Size   ID
 1   00   1    1    0   63    32   477      32  978912   83
 2   00   0    1  478   63    32   509  978944   65536   82
 3   00   0    0    0    0     0     0       0       0   00
 4   00   0    0    0    0     0     0       0       0   00
```

Vidíme, že první oblast začíná na cylindru 0, hlavičce 1 a sektoru 1 a končí na cylindru 477, sektoru 32 a hlavičce 63. Protože na stopě je 32 sektorů a disk má 64 hlaviček, tato oblast leží na celých prvních 478 cylindrech disku[1]. Program `fdisk` implicitně zarovnává oblasti na hranice celých cylindrů. Začíná na nejvnějším cylindru (0) a pokračuje směrem ke středu disku až na cylindr 477. Druhá oblast, odkládací oblast, začíná na následujícím cylindru (478) a pokračuje až k nejvnitřnímu cylindru disku.



Obrázek 8.3
Seznam disků

V době inicializace Linux zmapuje topologii pevných disků v systému. Zjistí, kolik disků a jakého typu systém obsahuje. Dále Linux zjistí, jak jsou jednotlivé disky rozděleny na oblasti. Všechny tyto údaje se ukládají jako seznam datových struktur `gendisk`, na nějž ukazuje ukazatel `gendisk_head`. Když se inicializují jednotlivé diskové subsystémy, například IDE, generují se datové struktury `gendisk` reprezentující nalezené disky. Děje se to ve stejné době, kdy ovladač registruje souborové operace a přidává záznam o sobě do struktury `blk_dev`. Každá datová struktura `gendisk` má jednoznačné hlavní číslo zařízení, které odpovídá hlavnímu číslu blokového zařízení. Například subsystém SCSI vytvoří jednu položku `gendisk` („sd") s hlavním číslem 8, hlavním číslem všech diskových zařízení SCSI. Na obrázku 8.3 vidíme dvě položky `gendisk`, první pro subsystém SCSI a druhou pro disk IDE. To je položka `ide0`, primární disk IDE.

[1] Pozn. překl.: Součástí první oblasti není pouze sektor 1 na 0. cylindru a 0. povrchu, tento sektor (úplně první sektor disku) je takzvaným *master zaváděcím sektorem* a je na něm (mimo jiné) uložena právě rozdělovací tabulka disku.

I když diskové subsystémy při své inicializaci budují struktury `gendisk`, ke hledání oblastí je používá pouze Linux. Každý diskový subsystém si navíc vytváří své vlastní struktury, které mu slouží k mapování hlavního a vedlejšího čísla zařízení na příslušné oblasti fyzických disků. Vždy, když se zapisuje nebo čte na nebo z blokového zařízení, ať už přes buffery nebo souborovou operací, jádro směruje operaci na příslušný ovladač pomocí hlavního čísla zařízení, které nalezne v souboru ovladače zařízení (například */dev/sda2*). Až samotný ovladač disku nebo diskový subsystém pak mapuje vedlejší číslo zařízení na konkrétní fyzický disk a oblastí.

8.5.1 Disky IDE

Na linuxových systémech se dnes nejčastěji používají disky IDE (Integrated Disk Electronic). IDE je diskové rozhraní, v podstatě vstupně/výstupní sběrnice podobně jako SCSI. Každý řadič IDE může ovládat dva disky, jeden v režimu *master*, druhý v režimu *slave*. Rozlišení disku master a slave se obvykle provádí přepínači přímo na disku. První řadič IDE v systému se označuje jako primární, druhý jako sekundární a tak dále. IDE zvládá datový přenos rychlostí kolem 3,3 MB za sekundu, maximální velikost disku IDE může být 538 MB. Extended IDE, (EIDE) povoluje disky o velikosti až 8,6 GB a přenosová rychlost se zvýšila na 16,6 MB za sekundu. Disky IDE a EIDE jsou levnější než disky SCSI a většina moderních PC má integrován jeden nebo více řadičů IDE.

Linux disky IDE pojmenovává v tom pořadí, v jakém nalezne jejich řadiče. Disk master na primárním řadiči se označuje jako `/dev/hda`, disk slave bude `/dev/hdb`. `/dev/hdc` je disk master na sekundárním řadiči IDE. Subsystém IDE registruje v jádře řadiče IDE, *ne* disky. Hlavní číslo primárního řadiče IDE je 3, hlavní číslo sekundárního řadiče je 22. Znamená to, že pokud má systém dva řadiče IDE, budou ve vektorech `blk_dev` a `blkdevs` na indexech 3 a 22 položky subsystému IDE. Soubory zařízení disků IDE toto označení reflektují, takže soubory `/dev/hda` i `/dev/hdb` budou mít jako hlavní číslo zařízení uvedeno 3. Všechny souborové nebo bufferové operace nad těmito blokovými zařízeními se předají subsystému IDE, protože jádro používá jako index hlavní číslo zařízení. Po vznesení požadavku už je starostí subsystému IDE aby zjistil, kterého disku se požadavek týká. K tomu používá IDE subsystém vedlejší číslo zařízení, které obsahuje informace potřebné ke směrování požadavku na správnou partici správného disku. Identifikátory zařízení pro `/dev/hdb`, disk slave na primárním řadiči IDE, jsou *(3,64)*. Identifikátor první oblast tohoto disku (`/dev/hdb1`) je *(3,65)*.

8.5.2 Inicializace subsystému IDE

Disky IDE existují prakticky po celou dobu existence počítačů IBM PC. V průběhu této doby se jejich rozhraní měnilo, takže inicializace subsystému IDE je složitější, než to vypadá na první pohled.

Linux je schopen podporovat maximálně čtyři řadiče IDE. Každý řadič je reprezentován strukturou `ide_hwif_t` ve vektoru `ide_hwifs`. Každá struktura `ide_hwif_t` obsahuje dvě struktury `ide_drive_t`, pro případný disk master a slave u řadiče. V době inicializace subsystému IDE Linux nejprve zjišťuje, zda informace o discích nejsou přístupny v paměti CMOS. Tato baterií zálohovaná paměť zachovává svůj obsah i po vypnutí počítače. Paměť se fyzicky nachází v hodinách reálného času, které běží bez ohledu na to, zda počítač je či není zapnutý. Obsah paměti CMOS je nastavován BIOSem a může Linuxu říci, jaké disky a řadiče byly nalezeny. Linux převezme od BIOSu údaje o geometrii disků a uloží je do datové struktury `ide_hwif_t`. Většina moderních PC používá chipset PCI, například Intel 82430 VX, který obsahuje i řadič PCI EIDE. Subsystém IDE používá volání PCI BIOSu k nalezení řadičů PCI (E)IDE v systému. Pak volá specifické rutiny daného chipsetu.

Jakmile je nalezeno rozhraní IDE či řadič, nastaví se jeho struktura `ide_hwif_t` tak, aby obsahovala informace o discích, připojených k tomuto řadiči. V době práce zapisuje ovladač příkazy IDE do příkazového registru řadiče IDE, který je k dispozici ve V/V adresovém prostoru. Implicitní V/V adresa pro řídící a stavové registry primárního řadiče IDE je v rozsahu *0x1F0 - 0x1F7*. Tyto adresy jsou dány konvencí už od dob prvních PC. Ovladač IDE registruje všechny řadiče ve vyrovnávací paměti bufferů a ve VFS zápisem do vektorů `blk_dev` a `blkdevs`. Ovladač IDE musí rovněž požádat o převzetí příslušného přerušení. Hodnoty přerušení jsou rovněž dány konvencí a je to 14 pro primární řadič IDE a 15 pro sekundární řadič. Stejně jako další konfigurační podrobnosti je však možno tyto hodnoty změnit. Pro každý řadič IDE nalezený v době startu systému přidává dále ovladač záznam IDE `gendisk` do seznamu těchto struktur. Tento seznam se později používá k nalezení rozdělovacích tabulek všech nalezených pevných disků. Kód pro kontrolu partic ví, že každý řadič IDE může mít připojeny dva disky.

8.5.3 SCSI disky

Sběrnice SCSI (Small Computer System Interface) je výkonná datová sběrnice, která podporuje až osm zařízení na jedné sběrnici včetně jednoho nebo více hostitelů. Každé zařízení musí mít jednoznačné identifikační číslo, které se obvykle nastavuje přepínači přímo na zařízení. Data je možno přenášet synchronně nebo asynchronně mezi libovolnými dvěma zařízeními na sběrnici s šířkou přenosu 32 bitů a maximální přenosovou rychlostí 40 MB za sekundu. Sběrnice SCSI přenáší mezi zařízeními jak data, tak stavové informace. Každá transakce mezi *iniciátorem* a *cílem* se může dělit na osm oddělených fází. Aktuální fázi sběrnice SCSI je možno určit z pěti řídících signálů na sběrnici. Uveďme si přehled jednotlivých fází:

BUS FREE Žádné zařízení nemá řízení nad sběrnicí a momentálně neprobíhají žádné transakce.

ARBITRATION Zařízení SCSI se pokouší získat řízení nad sběrnicí. Na adresové piny vysílá své identifikační číslo. Řízení získá zařízení s nejvyšším identifikačním číslem.

SELECTION Když zařízení získá řízení nad sběrnicí, signalizuje nyní cílové zařízení, kterému chce vyslat příkaz. Na adresové piny vysílá identifikátor cílového zařízení.

RESELECTION V průběhu zpracování žádosti se mohou zařízení odpojit. Cíl požadavku pak opětně vybírá iniciátora. Tuto fázi nepodporují všechna zařízení SCSI.

COMMAND Iniciátor cíli předává 6, 10 nebo 12 bajtový příkaz.

DATA IN, DATA OUT V průběhu těchto fází dochází k výměně dat mezi iniciátorem a cílem.

STATUS Do této fáze se přechází po dokončení všech příkazů a cíl nyní může iniciátorovi poslat stavový bajt indikující úspěch či neúspěch operace.

MESSAGE IN, MESSAGE OUT Doplňující informace posílané mezi iniciátorem a cílem.

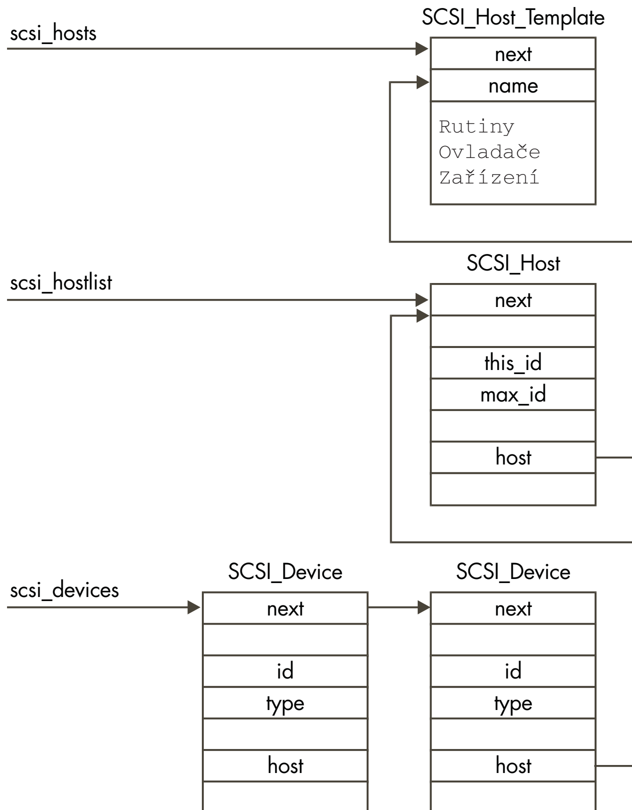
Subsystém SCSI v Linuxu se skládá ze dvou základních prvků, každý z nich je reprezentován svou datovou strukturou.

Hostitel Hostitel SCSI je fyzické hardwarové zařízení, řadič SCSI. Příkladem může být řadič NCR810 PCI SCSI. Pokud by v systému bylo více stejných řadičů SCSI, každý by byl reprezentován jako samostatný hostitel. Znamená to, že ovladač zařízení SCSI může řídit více než jeden řadič. Hostitelé SCSI ve většině případů fungují v roli iniciátora příkazu.

Zařízení Nejběžnějším zařízením SCSI je disk SCSI, nicméně standard SCSI podporuje i jiná zařízení, například pásky, CD-ROM a také obecná zařízení SCSI. Zařízení SCSI ve většině případů fungují jako cíle příkazů SCSI. K různým zařízením je nutno se chovat různě, například u zařízení s výměnnými médii jako jsou pásky a CD-ROM je nutno detekovat, zda je médium vloženo. Různé typy zařízení mají různá hlavní čísla zařízení, takže Linux může správně obsluhovat blokové požadavky.

Inicializace subsystému SCSI

Inicializace subsystému SCSI je poměrně složitá vzhledem k dynamické povaze sběrnic SCSI a jejich zařízení. Linux inicializuje zařízení SCSI v době zavádění, nejprve nalezne všechny řadiče SCSI a pak prohlíží jejich sběrnice a zjišťuje všechna připojená zařízení. Poté provede inicializaci jednotlivých zařízení a zpřístupní je zbytku jádra prostřednictvím normálních souborových a bufferových operací. Inicializace probíhá ve čtyřech fázích:



Obrázek 8.4
Datové struktury SCSI

Nejprve Linux zjistí, které hostitelské adaptéry SCSI, řadiče, jejichž ovladače byly při sestavování jádra zahrnuty v jádře, jsou hardwarově přítomny. Každý řadič SCSI má svou položku `Scsi_Host_Template` ve vektoru `builtin_scsi_hosts`. Datová struktura `Scsi_Host_Template` obsahuje ukazatele na rutiny poskytující služby specifické pro daný řadič, například detekci zařízení připojených ke sběrnici. Tyto rutiny volá subsystém SCSI v době své inicializace a jsou součástí ovladače zařízení SCSI daného typu. Každému detekovanému řadiči SCSI, tedy tomu, který je k systému skutečně připojen, se struktura

Scsi_Host_Template přidá do seznamu aktivních řadičů SCSI scsi_hosts. Každý detekovaný řadič je dále reprezentován strukturou Scsi_Host v seznamu scsi_hostlist. Například systém se dvěma řadiči NCR810 PCI SCSI vytvoří dvě položky Scsi_Host, jednu pro každý řadič. Každá struktura Scsi_Host ukazuje na strukturu Scsi_Host_Template, která reprezentuje ovladač zařízení.

V této fázi jsou tedy známy všechny řadiče SCSI. Subsystém SCSI musí dále zjistit, jaká zařízení jsou připojena ke sběrnicím jednotlivých řadičů. Zařízení SCSI se číslují čísly 0 až 7 včetně, číslo každého zařízení (identifikátor SCSI) musí být na sběrnici jedinečné. Identifikátor se obvykle nastavuje přepínači přímo na zařízení. Inicializační kód nalezne zařízení SCSI na sběrnici tím, že mu posílá příkaz TEST_UNIT_READY. Když zařízení odpoví, přečte se jeho identifikace zasláním příkazu ENQUIRY. Takto Linux získá jméno výrobce, označení modelu a verze. Příkazy SCSI jsou reprezentovány datovou strukturou Scsi_Cmnd a předávají se ovladači příslušného řadiče SCSI voláním rutin ovladače, jejichž adresy jsou uvedeny v jeho struktuře Scsi_Host_Template. Každé nalezené zařízení SCSI se reprezentuje strukturou Scsi_Device, která vždy ukazuje na svůj rodičovský Scsi_Host. Všechny struktury Scsi_Device se přidávají do seznamu scsi_devices. Na obrázku 8.4 vidíme, jak spolu hlavní datové struktury souvisejí.

Existují čtyři typy zařízení SCSI: disky, pásky, CD a obecná zařízení. Každý z těchto typů se v jádře registruje zvlášť jako blokové zařízení s rozdílným hlavním číslem. Registrují se ovšem pouze v případě, že je alespoň jedno zařízení daného typu připojeno. U každého typu, řekněme u disků, se udržuje samostatná tabulka zařízení. Tato tabulka slouží ke směrování blokových operací jádra na správný ovladač zařízení a SCSI řadič. Každý typ zařízení SCSI je reprezentován datovou strukturou Scsi_Device_Template. Ta obsahuje informace o typu zařízení a adresy rutin, které zajišťují různé operace. Subsystém SCSI používá tyto struktury k volání rutin příslušného typu zařízení pro každé zařízení SCSI. Jinak řečeno, pokud chce SCSI subsystém pracovat s diskem SCSI, bude volat rutiny pro typ diskové zařízení. Datové struktury Scsi_Type_Template se přidávají do seznamu scsi_devicelist pokud bylo detekováno jedno nebo více zařízení daného typu.

V závěrečné fázi inicializace subsystému SCSI se volají dokončovací rutiny pro každou registrovanou strukturu Scsi_Device_Template. U diskových typů dojde k roztočení všech disků a k zjištění jejich geometrie. Dále se přidává datová struktura gendisk reprezentující všechny disky SCSI do seznamu disků, který jsme viděli na obrázku 8.3.

Doručení požadavků blokovým zařízením

Jakmile proběhne inicializace subsystému SCSI, je možno zařízení SCSI používat. Každý aktivní typ zařízení SCSI se v jádře registruje, takže Linux na něj může směrovat požadavky na bloková zařízení. Může se jednat o bufferové operace přes struktury blk_dev nebo o soubo-

rové operace přes struktury blkdevs. Vezměme si jako příklad ovladač disku SCSI, na němž jsou jedna nebo více oblastí se souborovým systémem ext2. Jakým způsobem jádro zajistí předání požadavků na správný disk SCSI, na němž je připojená příslušná oblast ext2?

Každý požadavek na zápis nebo čtení bloku dat na nebo z diskové oblasti SCSI vede k vytvoření nové struktury request, která se přidává do seznamu current_request disků SCSI ve vektoru blk_dev. Pokud se seznam žádostí zpracovává, nemusí vyrovnávací paměť bufferů udělat nic dalšího, v opačném případě musí „postrčit" subsystém SCSI, aby začal požadavky ve frontě zpracovávat. Každý disk SCSI v systému je reprezentován strukturou Scsi_Disk. Ty se udržují ve vektoru rscsi_disk, který je indexován pomocí vedlejšího čísla zařízení oblastí disků SCSI. Například zařízení /dev/sdb1 má hlavní číslo 8 a vedlejší číslo 17, z něhož se generuje index 1. Každá datová struktura Scsi_Disk obsahuje ukazatel na strukturu Scsi_Device, která reprezentuje toto zařízení. Ta dále ukazuje na strukturu Scsi_Host, která je „vlastníkem" zařízení. Datová struktura request vytvořená vyrovnávací pamětí bufferů se přeloží na strukturu Scsi_Cmd, která udává, jaký příkaz se musí zařízení SCSI poslat a tento příkaz se uloží do fronty ve struktuře Scsi_Host, reprezentující příslušné zařízení. Tuto frontu zpracovává ovladač zařízení SCSI, který zajistí splnění požadavku na čtení nebo zápis.

8.6 Síťová zařízení

Síťová zařízení jsou, z pohledu síťového subsystému Linuxu, entity, které posílají a přijímají datové pakety. Obvykle se jedná o fyzická zařízení, například o síťovou kartu. Některá síťová zařízení jsou nicméně pouze logická zařízení, například zpětné zařízení, které slouží k posílání paketů sobě sama. Každé síťové zařízení je reprezentováno strukturou device. Ovladače síťových zařízení registrují svá zařízení v Linuxu v době inicializace sítě při zavádění jádra. Datová struktura device obsahuje informace o zařízení a adresy funkcí, které umožňují různým podporovaným síťovým protokolům použít služeb zařízení. Tyto funkce se většinou týkají odesílání dat prostřednictvím síťového zařízení. Zařízení používá standardní podpůrné mechanismy sítě k předání doručených dat příslušné protokolové vrstvě. Všechna síťová data (pakety), odeslaná i přijatá, jsou reprezentována strukturami sk_buff, což jsou pružné datové struktury, které umožňují snadno přidávat a odebírat hlavičky síťových protokolů. Mechanismy, jimiž protokoly různých síťových vrstev používají síťová zařízení a mechanismy, jimiž se data prostřednictvím struktur sk_buff předávají tam a zpět, jsou podrobně popsány v kapitole „Sítě". V této kapitole se soustředíme na datovou strukturu device a na detekci a inicializaci síťových zařízení.

Datová struktura device obsahuje následující informace o síťovém zařízení:

jméno	Na rozdíl od znakových a blokových zařízení, jejichž soubory zařízení se vytvářejí příkazem `mknod`, soubory síťových zařízení se vytvářejí samy při detekci a inicializaci síťových zařízení. Jména těchto souborů jsou standardní, každé z nich reprezentuje své zařízení. Více zařízení stejného typu se čísluje od nuly nahoru. Takže oblasti IDE disku se prezentují jako `/dev/hda1`, `/dev/hda2`, `/dev/hda3` a tak dále. Uveďme si jména některých běžných síťových zařízení:

`/dev/slN`	Zařízení SLIP
`/dev/pppN`	Zařízení PPP
`/dev/lo`	zpětná zařízení

sběrnicové informace Tyto informace ovladač zařízení potřebuje, aby mohl zařízení řídit. *irq* je číslo přerušení, které zařízení používá. *base address* je bázová adresa řídících a stavových registrů zařízení ve V/V adresovém prostoru. *DMA channel* je číslo kanálu DMA, který zařízení používá. Všechny tyto informace se nastavují při zavádění, kdy dochází k inicializaci zařízení.

příznaky rozhraní Tyto příznaky popisují vlastnosti a možnosti síťových zařízení:

IFF_UP	Rozhraní je aktivní a běží.
IFF_BROADCAST	Ve struktuře `device` je platná adresa broadcast.
IFF_DEBUG	Aktivován ladicí režim zařízení.
IFF_LOOPBACK	Jedná se o zpětné zařízení.
IFF_POINTTOPOINT	Jedná se o dvoubodové zařízení (PPP nebo SLIP).
IFF_NOTRAILERS	Zařízení nepodporuje trailery paketů.
IFF_RUNNING	Byly alokovány prostředky.
IFF_NOARP	Zařízení nepodporuje protokol ARP.
IFF_PROMISC	Zařízení je v promiskuitním režimu, přijímá všechny pakety bez ohledu na to, komu byly adresovány.
IFF_ALLMULTI	Zařízení přijímá všechny hromadně vysílané IP-rámce.
IFF_MULTICAST	Zařízení je schopno přijímat hromadně vysílané IP-rámce.

protokolové informace Popisují, jak může být toto zařízení použito různými protokolovými vrstvami:

mtu Maximální velikost paketu, který toto zařízení dokáže odeslat, v této hodnotě není započtena velikost hlavičky linkové vrstvy, kterou zařízení bude muset přidat. Podle této hodnoty volí síťové vrstvy, například IP, velikost odesílaných paketů.

family Tato hodnota udává, jakou protokolovou rodinu zařízení podporuje. U všech síťových zařízení v Linuxu je podporována rodina AF_INET, rodina internetových protokolů.

type Typ hardwarového rozhraní říká, k jakému typu média je zařízení připojeno. Síťová zařízení Linuxu mohou podporovat řadu různých fyzických médií, například Ethernet, X.25, Token Ring, SLIP, PPP a Apple Localtalk.

adresy Ve struktuře `device` jsou uložena čísla adres, které se vztahují k tomuto zařízení, včetně například IP adres.

fronta paketů Fronta struktur `sk_buff`, tedy paketů, které čekají na odeslání tímto zařízením.

podpůrné funkce Každé zařízení poskytuje standardní skupinu funkcí, které mohou protokolové vrstvy používat jako rozhraní k linkové vrstvě daného zařízení. Patří sem rutiny pro přípravu a odeslání rámce, rutiny pro připojení standardní hlavičky rámce a pro pořizování statistik. Statistiky zařízení je možno zjistit příkazem `ifconfig`.

8.6.1 Inicializace síťových zařízení

Ovladače síťových zařízení mohou být, stejně jako ovladače jiných zařízení, vestavěny přímo do jádra. Každé potenciální síťové zařízení je reprezentováno datovou strukturou `device` v seznamu síťových zařízení, na nějž ukazuje ukazatel `dev_base`. Pokud síťové vrstvy potřebují provést nějakou pro zařízení specifickou operaci, volají některou z řady síťových služebních rutin, jejichž adresy jsou uloženy ve struktuře `device`. Na počátku obsahuje každá datová struktura `device` pouze adresu inicializační rutiny.

S ovladači síťových zařízení je nutné řešit dva problémy. První problém je v tom, že ne pro každý ovladač vestavěný do jádra musí být skutečně přítomno síťové zařízení. Druhým problémem je, že ethernetová zařízení se vždy označují jako `eth0`, `eth1` a tak dále, bez ohledu na to, jaký ovladač fyzického zařízení se pro ně používá. Problém „chybějících" síťových zařízení se řeší velmi jednoduše. Když se volá inicializační rutina každého zařízení, vrátí

Průvodce jádrem operačního systému Linux

příznak, který říká, zda zařízení je či není přítomno. Pokud ovladač nenalezne žádné „své“ zařízení, odstraní se jeho položka `device` ze seznamu `dev_base`. Pokud ovladač nalezne zařízení, vyplní zbytek datové struktury `device` informacemi o zařízení a adresami podpůrných funkcí ovladače.

Druhý problém, problém dynamického mapování ethernetových zařízení `ethN`, se řeší elegantněji. V seznamu zařízení je standardně osm položek: `eth0`, `eth1` a tak dále až `eth7`. Inicializační rutina všech těchto zařízení je stejná, zkouší postupně všechny ovladače ethernetových karet v systému tak dlouho, dokud jeden z nich nenalezne své zařízení. Když ovladač nalezne zařízení, vyplní si svou strukturu `ethN`, kterou nyní vlastní. V této fázi ovladač zařízení rovněž provede fyzickou inicializaci zařízení a zjistí informace o použitém přerušení, DMA a podobně. Ovladač může nalézt několik instancí svého zařízení a v takovém případě převezme několik struktur `ethN`. Pokud se naplní všech osm standardních zařízení `ethN`, další ethernetová zařízení se nehledají.

Odkazy na zdrojové texty jádra

1 – `Viz fs/devices.c`

2 – `Viz include/-linux/major.h`

3 – `Viz ext2_read_inode() in fs/ext2/inode.c`

4 – `def_chr_fops`

5 – `Vizchrdev_open() in fs/devices.c`

6 – `Viz fs/devices.c`

7 – `Vizdrivers/block/-ll_rw_blk.c`

8 – `Viz include/-linux/blkdev.h`

9 – `Viz include/linux/-netdevice.h`

Kapitola

9

Souborový systém

V této kapitole popisujeme, jak Linux spravuje soubory na podporovaných souborových systémech. Popisujeme zde virtuální souborový systém (VFS) a vysvětlujeme dále podporu reálných souborových systémů.

Jednou z velkých výhod Linuxu je, že dokáže podporovat různé souborové systémy. Díky tomu je velmi pružný a dokáže spolupracovat s mnoha jinými operačními systémy. V době vzniku tohoto textu podporoval Linux 15 souborových systémů: `ext`, `ext2`, `xia`, `minix`, `umsdos`, `msdos`, `vfat`, `proc`, `smb`, `ncp`, `iso9660`, `sysv`, `hpfs`, `affs` a `ufs` a tento počet bezpochyby časem poroste.

V Linuxu, stejně jako v Unixu, se k jednotlivým samostatným souborovým systémům nepřistupuje prostřednictvím identifikátorů zařízení (jako jsou třeba čísla nebo označení diskových jednotek), namísto toho jsou všechny systémy složeny do jedné hierarchické stromové struktury, která reprezentuje celý souborový systém jako jedinou entitu. Při připojení každého souborového systému jej Linux do tohoto stromu přidá. Všechny souborové systémy libovolného typu se připojují do nějakého adresáře a soubory připojeného souborového systému překryjí původní obsah adresáře. Tento adresář se označuje jako připojovací adresář nebo také připojovací bod. Když se souborový systém odpojí, znovu se objeví původní obsah připojovacího adresáře.

Při inicializaci disku (například příkazem `fdisk`) se na něm vytváří struktura oblastí, které rozdělují jeden fyzický disk na několik logických částí. Každá oblast může nést jeden souborový systém, například `ext2`. Souborové systémy organizují soubory do logických hierarchických struktur pomocí adresářů, odkazů a podobně, všechno je uloženo v blocích na fyzickém zařízení. Souborový systém: Disková oblast `/dev/hda1` disku IDE (první oblast prvního disku IDE) je blokové zařízení. Zařízení, která mohou nést souborové systémy, se označují jako bloková zařízení. Souborové systémy v Linuxu požadují bloky uspořádané v prosté lineární ko-

lekci, nestarají se o fyzickou geometrii disku. Když souborový systém požaduje přečtení nějakého bloku, je úkolem ovladače zařízení, aby číslo bloku převedl na údaje, jimž zařízení rozumí: na stopy, sektory a povrchy. Souborový systém musí vypadat, chovat se a fungovat stejně bez ohledu na to, jaké fyzické zařízení jej hostí. Souborový systém nemusí dokonce být ani na lokálním disku, může se jednat o disk, vzdáleně připojený po síti. Podívejme se na následující příklad, kdy je kořenový souborový systém Linuxu umístěn na disku SCSI:

```
A       E       boot    etc     lib     opt             tmp     usr
C       F       cdrom   fd      proc    root            var     sbin
D       bin     dev     home    mnt     lost+found
```

Ani uživatelé, ani programy, které se soubory pracují, nepotřebují vědět, že adresář `/C` je ve skutečnosti připojený souborový systém VFAT na prvním disku IDE v systému. V tomto příkladu (je to souborový systém mého počítače) je `/E` master disk IDE na sekundárním řadiči IDE. Není vůbec důležité ani to, že primární řadič IDE je připojen sběrnicí PCI, sekundární je připojen sběrnicí ISA a ovládá také mechaniku CD-ROM. Pomocí modemu a síťového protokolu PPP se mohu připojit k síti v práci a v takovém případě si mohu souborový systém mého Linuxu v práci připojit do adresáře `/mnt/remote`.

Soubory v souborovém systému jsou kolekce dat. Text této kapitoly je například uložen v ASCII souboru pojmenovaném `filesystems.tex`. Souborový systém obsahuje jednak data uložená v souborech, ukládá si však i svou vlastní strukturu. Udržuje všechny informace, které uživatelé a procesy vnímají jako soubory, adresáře, odkazy, informace o přístupových právech a další. Navíc musí všechny tyto informace ukládat bezpečně, protože základní integrita celého operačního systému závisí právě na souborovém systému. Nikdo nebude používat souborový systém, kde se data a soubory náhodně ztrácejí.

`Minix`, první souborový systém používaný Linuxem, byl dost omezující a málo výkonný.

Názvy souborů mohly mít maximálně 14 znaků (což je ovšem pořád lepší než konvence 8.3) a velikost souborů byla omezena na 64 MB. Limit 64 MB vypadá sice na první pohled jako dostatečný, i skromné databáze však bývají větší. První souborový systém, vyvinutý speciálně pro Linux, byl Extended File System, `ext`, byl uveden v dubnu 1992 a i když řadu problémů vyřešil, jeho výkon byl stále nedostačující.

V roce 1993 se pak objevil Second Extended File System, `ext2`. Právě tento souborový systém si nyní podrobně popíšeme.

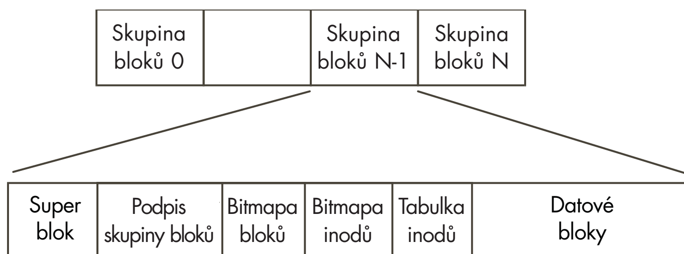
Když byl Linux doplněn o souborový systém `ext`, došlo ještě k další významné změně. Reálné souborové systémy byly odděleny od operačního systému a systémových služeb vrstvou, zvanou virtuální souborový systém, VFS.

VFS umožňuje podporu řady často velmi rozdílných souborových systémů, z nichž každý poskytuje na stranu VFS jednotné programové rozhraní. Všechny podrobnosti jednotlivých souborových systémů jsou softwarově zakryty, takže zbytku jádra Linuxu a všem programům v systému se všechny souborové systémy jeví stejně. Vrstva VFS umožňuje transparentně připojit mnoho různých souborových systémů najednou.

Virtuální souborový systém Linuxu je navržen tak, aby přístup k souborům byl co nejrychlejší a nejefektivnější. Je třeba dále zajistit, aby soubory a data byly uloženy správně. Tyto dva požadavky se mohou vzájemně vylučovat. VFS ukládá v paměti informace o každém připojeném a používaném souborovém systému. Je nutné věnovat velkou pozornost správné aktualizaci souborového systému, když dojde k modifikaci dat ve vyrovnávacích pamětech při vytváření, zápisu a rušení souborů a adresářů. Pokud byste mohli vidět datové struktury souborového systému v běžícím jádře, viděli byste zapisované a čtené bloky dat. Když ovladače zařízení pracují a načítají a zapisují data, stále dochází k vytváření a rušení datových struktur, reprezentujících adresáře a soubory. Nejdůležitější ze všech těchto vyrovnávacích pamětí je vyrovnávací paměť bufferů, která je integrována do mechanismů, jimiž jednotlivé souborové systémy přistupují ke svým blokovým zařízením. Když se přistupuje k jednotlivým blokům, ukládají se ve vyrovnávací paměti bufferů do různých front, které reprezentují jejich stav. Vyrovnávací paměť bufferů neuchovává pouze datové buffery, usnadňuje rovněž správu asynchronních rozhraní ovladačů blokových zařízení.

9.1 Souborový systém ext2

Souborový systém `ext2` byl navržen (Rémy Cardem) jako rozšiřitelný a výkonný souborový systém speciálně po Linux. Doposud je to nejoblíbenější souborový systém celé komunity Linuxu a je součástí všech dnešních distribucí Linuxu.



Obrázek 9.1

Fyzické rozvržení souborového systému ext2

Souborový systém `ext2`, stejně jako řada jiných souborových systémů, je vystavěn na předpokladu, že data ukládaná v souborech jsou uložena v datových blocích. Tyto datové bloky jsou stejně dlouhé a i když v různých souborových systémech ext2 mohou být délky odlišné,

nastavují se jednou provždy při vytváření souborového systému (příkazem `mke2fs`). Velikost každého souboru je zaokrouhlena nahoru na celý násobek velikosti bloku. Pokud má blok velikost 1 024 bajtů, pak bude soubor o délce 1 025 bajtů zabírat dva tyto bloky. Bohužel to znamená, že průměrně s každým souborem přicházíme o půl datového bloku. Linux ovšem v tomto případě, stejně jako většina jiných operačních systémů, dává přednost relativně neefektivnímu využití disku před větším zatížením CPU. Ne všechny bloky v souborovém systému slouží k ukládání dat, některé musejí obsahovat informace, popisující strukturu souborového systému. Systém `ext2` definuje topologii souborového systému popisem každého souboru prostřednictvím datové struktury *inode*. Inode udává, ve kterých blocích je soubor uložen, stejně jako přístupová práva k souboru, časy modifikace souboru a typ souboru. Každý soubor v souborovém systému `ext2` je popsán právě jedním inodem a každý inode má své jednoznačné identifikační číslo. Inody celého souborového systému jsou uloženy pohromadě v tabulce inodů. Adresáře v systému `ext2` jsou jen speciální soubory (rovněž popsané inodem), které obsahují ukazatele na inody popisující položky v daném adresáři.

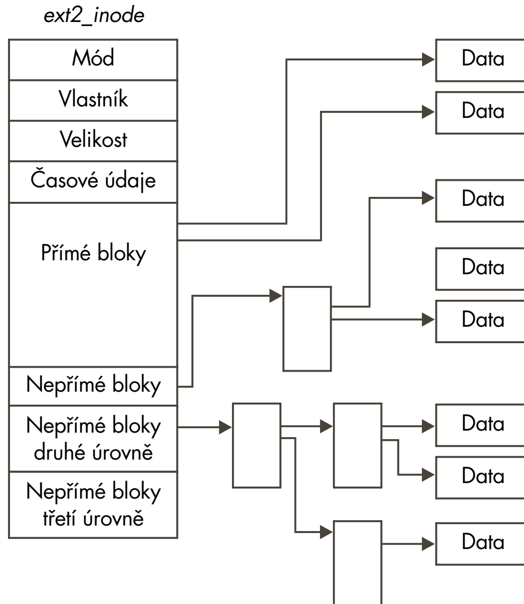
Na obrázku 9.1 vidíme strukturu, jakou systém `ext2` obsazuje bloky na blokově organizovaném zařízení. Z pohledu souborového systému představují bloková zařízení pouze lineární posloupnost bloků, které je možno číst a zapisovat. Souborový systém se nepotřebuje zabývat umístěním jednotlivých bloků na fyzickém médiu, to je starost ovladače zařízení. Kdykoliv souborový systém potřebuje přečíst informace nebo data z nějakého blokového zařízení, požádá příslušný ovladač zařízení o načtení celočíselného počtu bloků. Souborový systém ext2 rozděluje logickou oblast na níž je vytvořen na *skupiny bloků*.

Každá skupina duplikuje informace kritické pro integritu operačního systému a dále obsahuje skutečné soubory a adresáře jako bloky informací a dat. Tato duplikace je nezbytná pro případ havárie systému, kdy je nutná jeho obnova. V následujících částech je podrobněji vysvětlen obsah skupin bloků.

9.1.1 Inode

V souborovém systému `ext2` představuje inode základní stavební kámen: každý soubor a adresář v systému je popsán právě jedním inodem. Inody souborového systému `ext2` jsou pro každou skupinu bloků udržovány v tabulce inodů společně s bitovou mapou, která nese informace o alokovaných a nealokovaných inodech. Na obrázku 9.2 vidíme strukturu inode systému `ext2`. Kromě dalšího obsahuje inode následující informace:

mód Toto pole obsahuje dvě informace: co daný inode popisuje a přístupová práva, která uživatelé k danému objektu mají. V systému ext2 může inode popisovat soubor, adresář, symbolický odkaz, blokové zařízení, znakové zařízení nebo FIFO.



Obrázek 9.2
Inode systému ext2

informace o vlastníkovi Uživatelský a skupinový identifikátor vlastníka souboru nebo adresáře. Díky těmto údajům může souborový systém správně řídit přístup k objektu.

velikost Velikost souboru v bajtech.

časové značky Čas, kdy byl inode vytvořen a kdy byl naposledy modifikován.

datové bloky Ukazatel na blok dat obsahující data, která tento inode popisuje. Prvních dvanáct jsou ukazatelé na fyzické bloky obsahující data objektu, popsaného tímto inodem, poslední tři ukazatele zavádějí stále větší a větší nepřímo adresované objemy dat. Například ukazatel na dvojitý nepřímý ukazatel ukazuje na blok ukazatelů na bloky ukazatelů na datové bloky. Znamená to, že k souborům o délce dvanáct bloků a méně se přistupuje snáze a rychleji, než k souborům delším.

Je nutné si uvědomit, že inody systému `ext2` mohou popisovat rovněž speciální soubory zařízení. To nejsou skutečné soubory, ale odkazy, které mohou programy používat při přístupu

k zařízením. Všechny soubory zařízení v adresáři /dev slouží právě k tomu, aby programy mohly používat různá zařízení. Například program mount používá jako parametr jméno souboru zařízení, které se má připojit.

9.1.2 Superblok

Superblok obsahuje základní popis souborového systému. Informace uložené v superbloku používají mechanismy pro správu souborového systému při používání a údržbě systému. Obvykle se při připojení disku čte pouze superblok ve skupině bloků 0, nicméně každá skupina bloků obsahuje kopii superbloku pro případ poškození souborového systému. Mimo jiné obsahuje superblok následující informace.

kouzelné číslo (magic number)
Podle této hodnoty připojovací program pozná, že se jedná o superblok souborového systému ext2. V současné verzi se používá číslo *0xEF53*.

číslo revize Podle hlavního a vedlejšího revizního čísla připojovací software pozná, zda se jedná o verzi, která podporuje či nepodporuje určité funkce, dostupné pouze v určitých revizích systému. Součástí těchto údajů je i údaj o kompatibilitě, říkající, které funkce je možno v této verzi bezpečně používat.

připojovací počitadlo a maximální připojovací počet
Tyto dvě hodnoty slouží ke zjištění, zda se má provést úplná kontrola souborového systému. Připojovací počitadlo se inkrementuje při každém připojení systému a když dosáhne maximálního připojovacího počtu, objeví se hlášení „byl dosažen maximální počet připojení, spusťte e2fsck“.

číslo skupiny bloků Číslo skupiny bloků, v němž je tato kopie superbloku uložena.

velikost bloku Velikost bloku v této konfiguraci systému, například 1 024 bajtů.

počet bloků ve skupině Počet bloků ve skupině. Stejně jako u velikosti bloku, i tato hodnota se pevně nastavuje při vytváření systému.

počet volných bloků Počet volných bloků v souborovém systému.

počet volných inodů Počet volných inodů v souborovém systému.

první inode Číslo prvního inode souborového systému. První inode v systému ext2 je inode kořenového adresáře systému.

9.1.3 Deskriptor skupiny

Každá skupina bloků má svůj deskriptor, který ji popisuje. Stejně jako superblok, všechny deskriptory skupin jsou duplikovány ve všech skupinách pro případ poškození souborového systému.

4

Každý deskriptor skupiny obsahuje následující údaje:

bloková bitmapa Číslo bloku, v němž je uložena bloková bitová mapa této skupiny. Používá se při alokaci a dealokaci bloků.

inodová bitmapa Číslo bloku, v němž je uložena bitová mapa pro alokování inodů. Používá se při alokaci a dealokaci inodů.

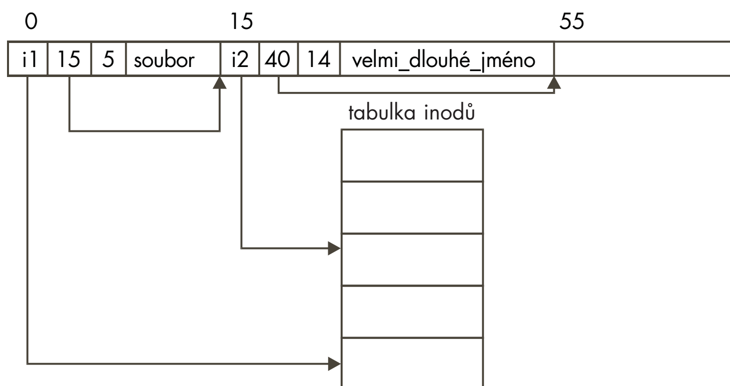
tabulka inodů Číslo počátečního bloku tabulky inodů této skupiny. Každý inode je reprezentován dříve popsanou datovou strukturou.

počet volných bloků, počet volných inode, počet vytvořených adresářů
Deskriptory skupin jsou uloženy jeden za druhým a dohromady vytvářejí tabulku deskriptorů skupin. Každá skupina obsahuje celou tabulku deskriptorů uloženou hned za kopií superbloku. Souborový systém `ext2` fakticky používá pouze první kopii (ve skupině 0). Ostatní kopie, stejně jako kopie superbloku, slouží pouze v případě porušení hlavní kopie.

9.1.4 Adresáře

V souborovém systému `ext2` jsou adresáře reprezentovány jako speciální soubory, které slouží k vytváření a ukládání přístupových cest k souborům v systému. Na obrázku 9.3 je vidět struktura adresářového záznamu v paměti.

5



Obrázek 9.3
Adresáře systému ext2

Adresářový soubor je seznam položek adresáře, z nichž každá obsahuje následující informace:

inode Inode této adresářové položky. Slouží jako index do pole inodů udržovaných v tabulce inodů skupiny bloků. Na obrázku 9.3 se adresářová položka pro soubor pojmenovaný `soubor` odkazuje na inode číslo `i1`.

délka jména Délka jména této adresářové položky v bajtech.

jméno Jméno této adresářové položky.

První dvě položky každého adresáře jsou vždy standardně „`.`“ a „`..`“ s významem „tento adresář“ a „rodičovský adresář“.

9.1.5 Nalezení souboru v systému ext2

Jména souborů v Linuxu používají stejný formát jako jména v systémech Unix. Jedná se o sérii jmen adresářů oddělených normálním lomítkem („/“) ukončená jménem souboru. Příkladem jména může být `/home/rusling/.cshrc`, kde `/home` a `/rusling` jsou jména adresářů a soubor se jmenuje `.cshrc`. Stejně jako ostatní systémy Unix se ani Linux nezajímá o formát samotného jména souboru, které může mít libovolnou délku a může se skládat z jakýchkoliv tisknutelných znaků. Když se v systému `ext2` hledá inode reprezentující určitý soubor, musí systém rozebrat jméno souboru na jednotlivé adresáře až dojde k samotnému souboru.

První potřebný inode je inode kořenového adresáře souborového systému a jeho číslo najdeme v superbloku souborového systému. Samotný inode pak nalezneme v tabulce inodů příslušné skupiny bloků. Pokud například kořen systému má inode číslo 42, musíme nalézt 42. inode z tabulky inodů nulté skupiny. Kořenový inode je inode adresáře, tedy mód tohoto inodu říká, že se jedná o adresář, a jeho datové bloky pak obsahují adresářové položky.

`home` je jen jedna z mnoha adresářových položek a tato adresářová položka nám řekne číslo inode, který popisuje adresář `/home`. Pak musíme načíst tento adresář (nejprve načteme jeho inode a poté adresářové položky z datových bloků, určených inodem) a hledáme položku `rusling`, z níž zjistíme číslo inodu popisujícího adresář `/home/rusling`. Nakonec přečteme adresářové položky určené inodem popisujícím adresář `/home/rusling` a budeme hledat číslo inode pro soubor `.cshrc` odkud se už dostaneme k datovým blokům, obsahujícím informace uložené v tomto souboru.

9.1.6 Změna velikosti souboru v systému ext2

Běžným problémem souborových systémů je jejich sklon k fragmentaci. Bloky obsahující data souboru jsou rozmístěny v celém souborovém systému a sekvenční přístup k souboru se stává tím více neefektivní, čím je rozptyl bloků větší. Souborový systém `ext2` se snaží tomuto problému předcházet tak, že nové bloky pro soubor alokuje fyzicky blízko ke stávajícím blokům, přinejmenším však ve stejné skupině bloků. Pouze pokud se mu to nezdaří, alokuje bloky z jiné skupiny bloků.

Vždy když se proces pokouší o zápis dat do souboru, souborový systém kontroluje, zda data nepřesahují za poslední blok alokovaný danému souboru. Pokud ano, pak je nutné pro soubor alokovat další blok. Dokud alokace neskončí, proces nemůže pokračovat - před pokračováním musí počkat, dokud souborový systém neprovede alokaci bloku a nezapíše do něj data. První věc, kterou alokační rutina bloku udělá, je, že zamkne superblok souborového systému. Alokace a dealokace bloků vede ke změnám údajů v superbloku a souborový systém nemůže dopustit, aby tyto změny provádělo více procesů najednou. Pokud alokaci dalšího bloku požaduje i jiný proces, musí počkat, dokud neskončí alokace pro předchozí proces. Procesy čekající na superblok jsou pozastaveny a nemohou pracovat dokud superblok nebude opět uvolněn. Přístup k superbloku se přiděluje na principu „kdo dřív přijde, ten dřív mele“ a jakmile proces jednou získá superblok, vlastní jej, dokud alokace neskončí. Po získání superbloku proces nejprve kontroluje, zda je v systému dostatek volných bloků. Pokud není dost volných bloků, požadavek na alokaci nemůže být splněn a proces se vzdává vlády nad superblokem.

Pokud v souborovém systému je dostatek volných bloků, proces se je pokusí alokovat.

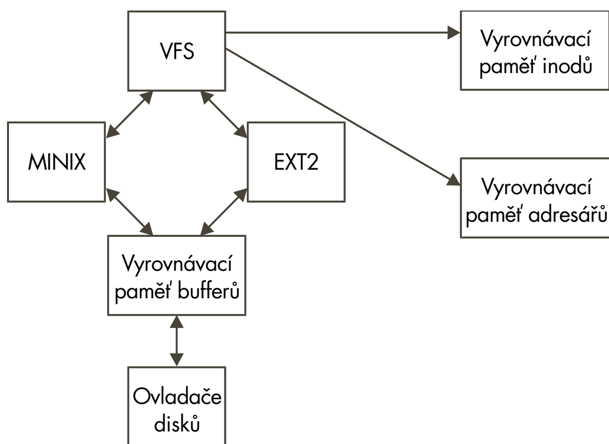
6 Pokud je souborový systém `ext2` nastaven tak, aby prováděl prealokaci bloků, můžeme použít jeden z prealokovaných bloků. Prealokovaný blok ve skutečnosti neexistuje, je pouze rezervován v bitové mapě alokovaných bloků. VFS inode, reprezentující soubor pro nějž se pokoušíme nový blok alokovat, obsahuje dvě položky specifické pro souborový systém `ext2`, `prealloc_block` a `prealloc_count`, což jsou čísla prvního prealokovaného bloku a jejich počet. Pokud není prealokován žádný blok nebo není prealokace bloků aktivována, musí souborový systém `ext2` provést alokaci nového bloku. Systém se nejprve pokusí alokovat blok bezprostředně za posledním alokovaným blokem souboru. Logicky je to to nejefektivnější místo, protože se tak výrazně urychluje sekvenční přístup. Pokud blok není volný, pak se hledá volný blok někde v rámci 64 následujících bloků. Takovýto blok, i když není ideální, je přinejmenším dostatečně blízko a ve stejné skupině bloků, jako ostatní bloky souboru.

Pokud není ani takovýto blok volný, pak se hledá ve všech ostatních skupinách bloků dokud se nepodaří nějakou volnou sekvenci bloků nalézt. Kód alokace bloků se pokouší ve skupinách bloků nalézt sérii osmi volných bloků. Pokud nenalezne osm volných bloků pohromadě, spokojí se i s méně. Pokud byla aktivována a použita prealokace bloků, provede se aktualizace údajů `prealloc_block` a `prealloc_count`.

Ať už bude volný blok nalezen kdekoliv, provede alokační kód aktualizaci bitové mapy alokovaných bloků příslušné skupiny bloků a příslušného datového bufferu ve vyrovnávací paměti bufferů. Datový buffer je jednoznačně identifikován identifikátorem příslušného zařízení souborového systému a číslem alokovaného bloku. Datový buffer se vynuluje a označí jako modifikovaný, aby bylo zřejmé, že jeho obsah nebyl doposud zapsán na fyzický disk. Nakonec se jako modifikovaný označí superblok, aby se vědělo, že byl změněn, a odemkne se. Pokud existují další procesy čekající na superblok, spustí se první z nich ve frontě a získá výhradní právo manipulace se superblokem pro své souborové operace. Data procesu se zapíší do nového datového bloku a až dojde k zaplnění i tohoto bloku, celý postup se opakuje a alokuje se nový datový blok.

9.2 Virtuální souborový systém (VFS)

Obrázek 9.4 znázorňuje vztah mezi virtuálním souborovým systémem jádra Linuxu a reálnými souborovými systémy. Virtuální souborový systém musí spravovat všechny souborové systémy připojené v daném okamžiku. K tomu účelu používá datové struktury popisující celý (virtuální) souborový systém a také reálné, připojené souborové systémy.



Obrázek 9.4
Logický diagram virtuálního souborového systému

7

Je poněkud zavádějící, že VFS popisuje souborové systémy pomocí superbloků a inodů prakticky stejným způsobem, jako je používá souborový systém `ext2`. Stejně jako inody v `ext2`, i inody systému VFS popisují soubory a adresáře v tomto systému, obsah a topologii virtuálního souborového systému. Od této chvíle budu, aby se předešlo zmatení pojmů, používat termíny inode VFS a superblok VFS k odlišení od jejich jmenovců v systému `ext2`.

Při inicializaci se každý souborový systém registruje ve VFS. Dochází k tomu v době, kdy probíhá inicializace samotného operačního systému při zavádění. Reálné souborové systémy jsou buď vestavěny přímo v jádře, nebo se dohrávají jako samostatné moduly. Moduly souborových systémů se nahrávají podle potřeby, takže napříkald pokud je jako modul jádra implementován souborový systém `vfat`, nahraje se pouze v případě, že se připojí souborový systém `vfat`. Když se připojí souborový systém na blokovém zařízení, včetně kořenového souborového systému, musí VFS přečíst jeho superblok. Rutina pro načtení superbloku každého typu souborového systému musí zjistit topologii systému a namapovat tyto informace do superbloku VFS. VFS udržuje seznam připojených souborových systémů společně s jejich superbloky VFS. Každý superblok VFS obsahuje informace a ukazatele na rutiny, provádějící jednotlivé funkce. Takže například superblok reprezentující připojený systém `ext2` obsahuje ukazatel na rutinu načtení inode specifickou pro systém `ext2`. Rutina pro načtení `ext2` inodu, stejně jako rutiny pro načtení inodu jiných souborových systémů, vyplňuje údaje v inodu VFS. Každý superblok VFS obsahuje ukazatel na kořenový inode VFS příslušného souborového systému. Pro kořenový souborový systém je to inode reprezentující adresář „/“. Takovéto mapování informací je velmi efektivní pro souborový systém `ext2`, je však méně výkonné pro jiné souborové systémy.

Když procesy v systému přistupují k souborům a adresářům, volají se systémové rutiny, které procházejí inody VFS.

Například zadání příkazu `ls` pro adresář nebo `cat` pro soubor způsobí, že VFS prohledá své inody VFS, reprezentující celý souborový systém. Protože každý soubor a adresář v systému je reprezentován inodem VFS, často se opakuje přístup k určitým inodům. Tyto inody jsou uloženy ve vyrovnávací paměti inodů, takže se přístup k nim urychluje. Pokud nějaký inode není ve vyrovnávací paměti, pak se musí zavolat rutina pro konkrétní souborový systém, která zajistí načtení příslušného inode. Operace načtení inode způsobí, že inode se umístí ve vyrovnávací paměti a další přístupy k tomuto inodu už končí jen ve vyrovnávací paměti. Nejméně používané VFS inody se z vyrovnávací paměti odstraňují.

Všechny souborové systémy v Linuxu používají společnou vyrovnávací paměť bufferů, ve které se ukládají datové buffery fyzických zařízení kvůli zrychlení přístupu k fyzickým zařízením jednotlivých souborových systémů.

Tato vyrovnávací paměť bufferů je na souborovém systému nezávislá a je integrována v mechanismu, který jádro Linuxu používá k alokaci, čtení a zápisu datových bufferů. Je velmi výhodné, že souborové systémy Linuxu jsou nezávislé na příslušném fyzickém zařízení a na ovladači, který s tímto zařízením pracuje. Všechna bloková zařízení se registrují v jádře Linuxu a nabízejí uniformní, blokové, obvykle asynchronní rozhraní. Dělají to i poměrně komplikovaná bloková zařízení, jako jsou například disky SCSI. Protože reálné souborové systémy načítají data z příslušných fyzických zařízení, vede to ke generování požadavků na ovladač

blokového zařízení, který musí zajistit načtení bloku ze svého zařízení. Do rozhraní blokového zařízení je integrována vyrovnávací paměť bloků. Když se různými souborovými systémy načítají bloky, ukládají se v globální vyrovnávací paměti bufferů sdílené všemi souborovými systémy a jádrem Linuxu. Buffery v této paměti jsou identifikovány číslem bloku a jednoznačným identifikátorem zařízení, z něhož byly načteny. Takže pokud se nějaká data vyžadují častěji, získají se přímo z vyrovnávací paměti a ne z disku, což by trvalo déle. Některá zařízení podporují dopředné čtení, kdy se datové bloky načítají spekulativně čistě pro případ, že by byly zapotřebí.

10 VFS dále udržuje vyrovnávací paměť prohledávaných adresářů, takže se rychle naleznou inody často používaných adresářů.

Můžete jako malý pokus vyzkoušet vypsat obsah adresáře, který jste si doposud neprohlíželi. Při prvním výpisu zaregistrujete určité zpoždění, druhý výpis však proběhne okamžitě. Vyrovnávací paměť adresářů neobsahuje přímo inody adresářů, ty jsou uloženy ve vyrovnávací paměti inodů, obsahuje pouze mapování mezi plným jménem adresáře a číslem jeho inodu.

9.2.1 Superblok VFS

Každý připojený souborový systém je reprezentován superblokem VFS. Kromě jiného obsahuje superblok VFS následující informace:

11

zařízení Identifikátor blokového zařízení, na němž je souborový systém uložen. Například `/dev/hda1`, první disk IDE v systému, má identifikátor *0x301*.

ukazatele inodů Ukazatel `mounted` ukazuje na kořenový inode tohoto souborového systému. Ukazatel `covered` ukazuje na inode reprezentující adresář, na němž je tento souborový systém připojen. Superblok kořenového souborového systému neobsahuje ukazatel `covered`.

velikost bloku Velikost bloku v daném souborovém systému, například 1 024 bajtů.

operace superbloku Ukazatel na množinu rutin superbloku daného souborového systému. Tyto rutiny mimo jiné slouží ke čtení a zápisu inodů a superbloků.

typ souborového systému
Ukazatel na datovou strukturu `file_system_type` připojeného systému.

specifické pro souborový systém
Ukazatel na informace, používané konkrétním souborovým systémem.

9.2.2 Inode VFS

Stejně jako v systému `ext2`, i ve VFS je každý soubor, adresář a podobně reprezentován právě jedním inodem VFS.

Informace v jednotlivých inodech VFS se získávají z informací reálných souborových systémů prostřednictvím systémově závislých rutin. inody VFS existují pouze v paměti jádra a ukládají se ve vyrovnávací paměti inodů VFS tak dlouho, dokud jsou potřebné. Kromě jiného obsahují inody VFS následující informace:

12

zařízení Identifikátor zařízení, na němž je uložen soubor nebo jiný objekt, který tento inode VFS reprezentuje.

číslo inode Číslo inodu souboru, je jedinečné v rámci souborového systému. Kombinace hodnot `device` a `inode number` je jedinečná v celém VFS.

mód Stejně jako u `ext2` i zde tento údaj popisuje typ objektu reprezentovaného tímto inodem a přístupová práva k němu.

uživatelská id Identifikátory vlastníka.

časy Časy vytvoření, modifikace a zápisu.

velikost bloku Velikost bloku tohoto souboru v bajtech, například 1 024 bajtů.

operace Ukazatel na blok adres rutin. Tyto rutiny jsou závislé na souborovém systému a zajišťují operace nad tímto inodem, například zkrácení souboru reprezentovaného tímto inodem.

počitadlo Počet systémových komponent, které inode momentálně používají. Nulová hodnota znamená, že inode je možno zrušit nebo nově obsadit.

zámek Toto pole slouží k uzamčení inodu VFS, například v době, kdy je načítán ze souborového systému.

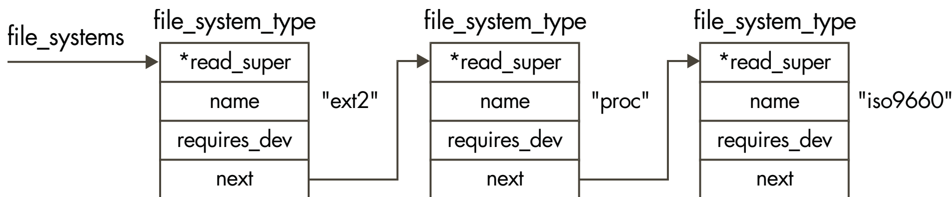
modifikován Příznak modifikace inode, pokud byl modifikován, bude nutná modifikace i ve fyzickém souborovém systému.

informace specifické pro souborový systém

9.2.3 Registrace souborových systémů

Když sestavujete jádro Linuxu, určujete, které z podporovaných souborových systémů budete chtít vestavět do jádra. Po sestavení jádra obsahuje inicializační kód souborového systému volání inicializačních rutin všech vestavěných souborových systémů.

13



Obrázek 9.5
Registrované souborové systémy

Souborové systémy mohou být v Linuxu sestaveny rovněž jako samostatné moduly a v takovém případě se provede jejich nahrání až v okamžiku, kdy budou zapotřebí, nebo když jsou nahrány ručně příkazem `insmod`. Vždy, když je nahrán modul souborového systému, registruje se v jádře a když se odstraňuje, ruší svou registraci. Inicializační rutina každého souborového systému provede registraci ve virtuálním souborovém systému, kde je systém reprezentován datovou strukturou `file_system_type`, která obsahuje jméno souborového systému a ukazatel na rutinu načtení jeho superbloku VFS. Na obrázku 9.5 vidíme, že struktury `file_system_type` se ukládají do seznamu, na jehož začátek ukazuje ukazatel `file_systems`. Každá datová struktura `file_system_type` obsahuje následující informace:

rutina načtení superbloku Tuto rutinu VFS volá, když je připojena nějaká instance souborového systému.

jméno souborového systému Jméno souborového systému, například `ext2`.

příznak potřebnosti zařízení Potřebuje tento souborový systém podpůrné zařízení? Ne všechny souborové systémy potřebují zařízení, na němž jsou uloženy. Například souborový systém `/proc` nepotřebuje blokové zařízení.

Registrované souborové systémy je možno zjistit v souboru `/proc/filesystems`. Např:

```
      ext2
nodev proc
      iso9660
```

9.2.4 Připojení souborového systému

Když superuživatel připojuje souborový systém, musí jádro Linuxu nejprve ověřit parametry předané systémovému volání. Přestože příkaz `mount` provádí nějaké základní kontroly, nemůže vědět, jaké souborové systémy jádro podporuje a zda požadovaný přípojný bod opravdu existuje. Vezměme například následující příkaz:

```
$ mount -t iso9660 -o ro /dev/cdrom /mnt/cdrom
```

Tento příkaz předává jádru tři informace: jméno souborového systému, fyzické blokové zařízení, které souborový systém obsahuje, a kam v topologii stávajícího souborového systému má být nový souborový systém připojen.

První věc, kterou musí VFS udělat, je, že musí nalézt příslušný souborový systém. Provede to prohlížením známých souborových systémů ve strukturách `file_system_type`, na něž ukazuje ukazatel `file_systems`.

15

16

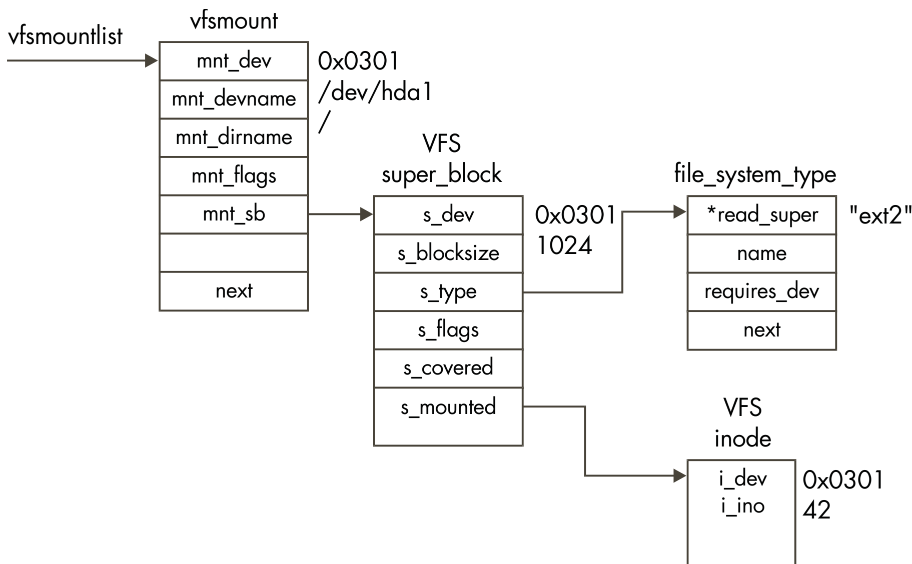
Pokud najde souborový systém požadovaného jména, ví, že daný typ souborového systému je jádrem podporován a má adresu systémově specifické rutiny pro načtení superbloku souborového systému. Pokud nenalezne souborový systém požadovaného jména, není nic ztraceno v případě, že jádro dokáže nahrávat moduly podle potřeby (viz kapitola „Moduly“). V takovém případě jádro požádá démona jádra o nahrání modulu příslušného souborového systému a poté může pokračovat.

Pokud požadované fyzické zařízení není dosud připojeno, musí se nalézt inode VFS pro adresář, který má být připojovacím bodem zařízení. Tento VFS inode může být ve vyrovnávací paměti inodů nebo jej bude nutné načíst z blokového zařízení, na němž je uložen souborový systém připojovacího bodu. Když se inode nalezne, zkontroluje se, zda je to opravdu adresář a zda na něj není připojen jiný souborový systém. Jeden adresář není možno použít jako připojovací bod pro více než jeden souborový systém.

V této fázi musí připojovací kód VFS alokovat superblok VFS a předat mu informace o rutině pro načtení superbloku připojovaného systému. Všechny superbloky VFS jsou udržovány ve vektoru `super_blocks` datových struktur `super_block` a při připojování se jedna z nich musí alokovat. Rutina načtení superbloku vyplní údaje superbloku VFS informacemi, které načte z fyzického zařízení. Pro souborový systém `ext2` je toto mapování nebo překlad informací velmi jednoduché, protože se jednoduše přečte superblok systému `ext2` a naplní se jím superblok VFS. U jiných souborových systémů, například systému MS-DOS, to není tak jednoduché. Ať už je souborový systém jakýkoliv, naplnění superbloku VFS předpokládá, že souborový systém musí z fyzického blokového zařízení, na němž je uložen, přečíst své charakteristiky. Pokud není možno ze zařízení číst nebo pokud neobsahuje požadovaný typ souborového systému, připojování skončí chybou.

Každý připojený souborový systém je popsán datovou strukturou `vfsmount`, viz obrázek 9.6. Tyto struktury jsou seřazeny v seznamu, na nějž ukazuje ukazatel `vfsmntlist`.

17



Obrázek 9.6
Připojený souborový systém

Další ukazatel (`vfsmnttail`) ukazuje na poslední položku tohoto seznamu a ukazatel `mru_vfsmnt` ukazuje na naposledy použitý souborový systém. Každá struktura `vfsmount` obsahuje číslo zařízení blokového zařízení, na němž je souborový systém uložen, adresář, do nějž je systém připojen, a ukazatel na superblok VFS alokovaný při připojení tohoto souborového systému. Superblok VFS pak ukazuje na datovou strukturu `file_system_type` tohoto souborového systému a na jeho kořenový inode. Tento inode zůstává rezidentně ve vyrovnávací paměti inodů VFS po celou dobu, kdy je souborový systém připojen.

9.2.5 Nalezení souboru ve virtuálním souborovém systému

Při hledání inodu VFS ve virtuálním souborovém systému musí VFS rozložit jméno na jednotlivé adresáře a postupně nalézt inody všech adresářů ve jméně. Hledání každého adresáře obnáší volání rutin konkrétního souborového systému, jejichž adresy jsou uloženy v inodu VFS reprezentujícím rodičovský adresář. Celé to funguje díky tomu, že kořenový adresář každého souborového systému máme vždy k dispozici a ukazuje na něj superblok VFS daného souborového systému. Vždy, když reálný souborový systém hledá adresář, pokouší se jej nejprve nalézt ve vyrovnávací paměti adresářů. Pokud v této paměti adresář není, získává reálný souborový systém VFS inode buď přímo ze souborového systému nebo z vyrovnávací paměti inodů.

9.2.6 Odpojení souborového systému

V příručce k mému autu je postup montáže obvykle popsán jako „opak demontáže“ a tato moudrost v zásadě platí i pro odpojení souborového systému.

Souborový systém není možno odpojit, pokud někdo v systému používá nějaký z jeho souborů. Nemůžete tedy například odpojit /mnt/cdrom, pokud nějaký proces používá tento adresář nebo některý z jemu podřízených adresářů. Pokud cokoliv používá souborový systém, který má být odpojen, mohou být ve vyrovnávací paměti inodů VFS nějaké inody VFS tohoto souborového systému. Kód odpojení tyto inody hledá tak, že prochází seznam inodů a hledá inody vlastněné zařízením, na němž je souborový systém umístěn. Pokud byl modifikován superblok VFS systému, musí se zapsat zpět na disk. Po jeho zapsání na disk se paměť zabraná superblokem vrátí zpět jádru. Nakonec se ze seznamu vfsmntlist odstraní struktura vfsmount a uvolní se z paměti.

18

19

9.2.7 Vyrovnávací paměť inodů VFS

Když se pracuje s připojenými systémy, průběžně dochází k načítání a případnému zápisu jejich inodů VFS. Virtuální souborový systém udržuje vyrovnávací paměť inodů, která slouží ke zrychlení přístupu na všechny připojené souborové systémy. Vždy, když se podaří načíst inode VFS z vyrovnávací paměti, ušetří se přístup na fyzické zařízení.

20

Vyrovnávací paměť inodů VFS je implementována jako hashovací tabulka, jejíž položky jsou ukazatelé na seznamy inodů VFS se stejnou hashovací hodnotou. Hashovací hodnota inodu se počítá z čísla inodu a z identifikátoru příslušného fyzického zařízení, který obsahuje daný souborový systém. Vždy, když VFS potřebuje přistupovat k inodu, nejprve se podívá do vyrovnávací paměti inodů. Při hledání inodu ve vyrovnávací paměti vypočítá systém nejprve jeho hashovací hodnotu a pak ji použije jako index do hashovací tabulky inodů. Tím získá ukazatel na seznam inodů se stejnou hashovací hodnotou. Z tohoto seznamu načítá každý inode dokud nenalezne ten, jehož číslo a zařízení odpovídá požadovanému inodu.

Pokud se podaří nalézt inode ve vyrovnávací paměti, inkrementuje se jeho počitadlo, aby se detekovalo, že inode má dalšího uživatele, a systém pokračuje v práci. Pokud inode nebyl ve vyrovnávací paměti, musí se nalézt volný inode VFS, aby souborový systém mohl načíst požadovaný inode do paměti. VFS má řadu možností jak získat volný inode. Pokud systém může alokovat další inode VFS, pak se to provede - alokuje se stránka jádra, vytvoří se v ní volné inody a připojí se k seznamu inodů VFS. Všechny inody VFS vytvářejí seznam, na nějž

ukazuje ukazatel `first_inode` a hashovací tabulka inodů. Pokud už má systém alokován maximální povolený počet inodů, musí nalézt inode, který bude vhodným kandidátem na nové použití. Dobrými kandidáty jsou inody s nulovým počitadlem uživatelů, což znamená, že je momentálně nikdo v systému nevyužívá. Významné inody VFS, například kořenové inody jednotlivých souborových systémů, mají počitadlo použití vždy nenulové, a tak nemohou být nikdy nahrazeny. Když se podaří nalézt vhodný inode pro nové použití, vyčistí se. Inode mohl být modifikován a v takovém případě je nutné zapsat jej zpět do souborového systému, nebo může být zamčený a systém musí před pokračováním práce počkat na jeho odemčení. Před novým použitím musí být inode vyčištěn.

Ať už je nový inode VFS nalezen jakkoliv, zavolá se rutina specifického souborového systému, která jej naplní informacemi ze skutečného souborového systému. V době plnění má inode počitadlo použití rovno jedné a je uzamčen, takže jej nikdo nemůže použít dříve, než bude inode obsahovat platné informace.

Aby se získal požadovaný inode VFS, bude systém možná muset postupně přistupovat k několika jiným inodům. K tomu dochází, když čtete adresář: je sice zapotřebí pouze inode požadovaného adresáře, musí se však načíst inody i mezilehlých adresářů. Jak se vyrovnávací paměť inodů plní a rozrůstá, méně používané inody se ruší a v paměti zůstávají pouze ty častěji používané.

9.2.8 Vyrovnávací paměť adresářů

Kvůli zrychlení přístupu k často používaným adresářům vytváří VFS vyrovnávací paměť adresářových položek.

Když reálný souborový systém prohledává adresáře, zjištěné informace se přidávají do vyrovnávací paměti adresářů. Při příštím prohlížení stejného adresáře, například při vypisování jeho obsahu nebo při otevírání souboru v tomto adresáři, se informace naleznou ve vyrovnávací paměti. Ve vyrovnávací paměti se ukládají pouze krátké adresářové položky (do délky 15 znaků), což je ale rozumné řešení, protože nejčastěji se používají právě nejkratší jména. Například adresář `/usr/X11R6/bin` je velmi často používán spuštěným X serverem.

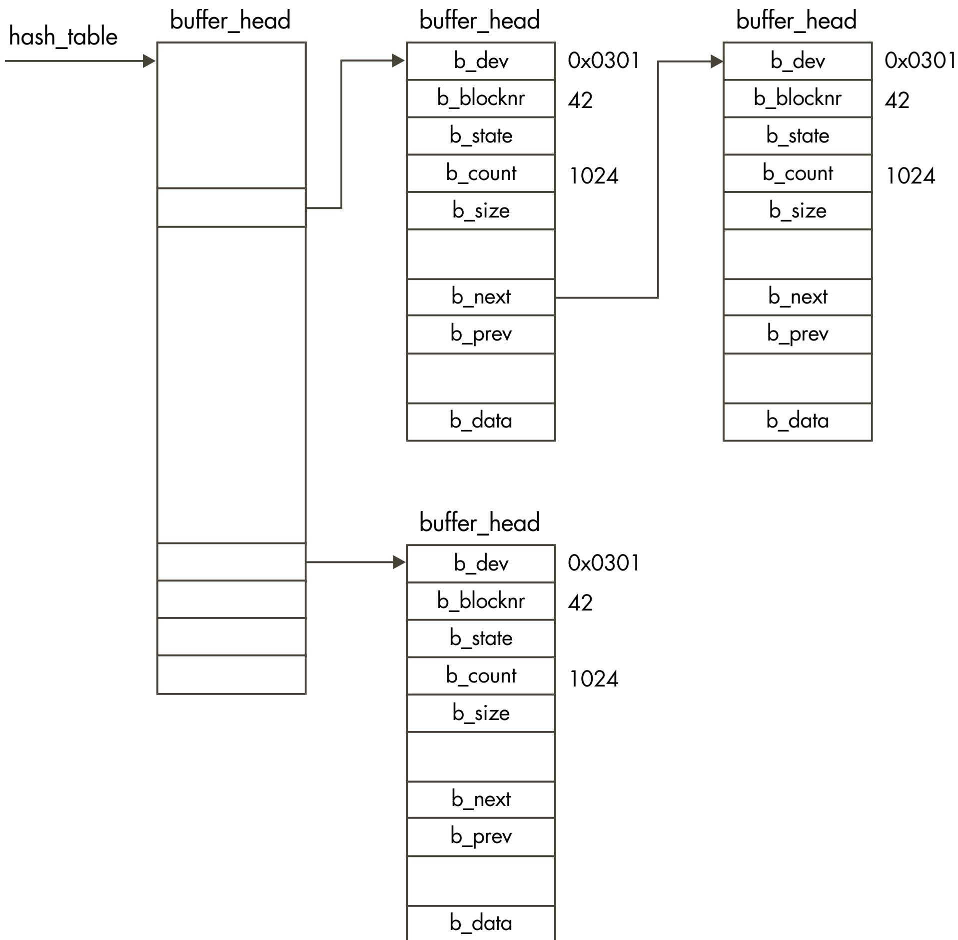
Vyrovnávací paměť adresářů se skládá z hashovací tabulky, jejíž každá položka ukazuje na seznam adresářových položek, které mají stejnou hashovací hodnotu. Hashovací funkce vypočítává offset v tabulce z čísla zařízení, na němž je příslušný souborový systém uložen, a ze jména adresáře. Díky tomu se jednotlivé položky dají rychle nalézt. Bylo by k ničemu mít vyrovnávací paměť, pokud by hledání (a případné nenalezení) položky v této paměti trvalo dlouho.

Aby byl obsah paměti platný a aktuální, udržuje VFS seznamy položek podle toho, kdy byly naposledy použity - takzvané LRU (Least Recently Used) seznamy. Vždy když se do vyrovnávací paměti přidává adresář, přidává se na konec LRU seznamu první úrovně. Pokud je vyrovnávací paměť plná, odstraní se tím zároveň položka na začátku tohoto LRU seznamu. Jakmile je položka použita znovu, přemístí se na konec LRU seznamu druhé úrovně. I zde může případně dojít ke zrušení položky ze začátku tohoto seznamu. Rušení položek ze začátků obou úrovní seznamů je zcela v pořádku. Jediný důvod, jak se mohla položka na začátku seznamu ocitnout, je ten, že už k ní delší dobu nebylo přistupováno. Pokud by se používala, byla by více ke konci seznamu. Položky v LRU seznamu druhé úrovně jsou uloženy „bezpečněji" než v seznamu první úrovně. To je pochopitelně záměrné, protože tyto položky byly nejen nalezeny, ale také se s nimi opakovaně pracovalo.

9.3 Vyrovnávací paměť bufferů

Při používání připojených souborových systémů se objevuje řada požadavků na načtení nebo zápis bloků na bloková zařízení. Všechny požadavky na čtení a zápis se ovladači zařízení předávají jako datová struktura `buffer_head` prostřednictvím volání standardních rutin jádra. Tato struktura obsahuje všechny informace, které ovladač blokového zařízení potřebuje - identifikátor blokového zařízení jednoznačně identifikuje zařízení a číslo bloku oznamuje ovladači, který blok má přečíst. Všechna bloková zařízení jsou chápána jako lineární posloupnost bloků stejné velikosti. Kvůli zrychlení přístupu k fyzickým blokovým zařízením používá Linux vyrovnávací paměť blokových bufferů. Všechny blokové buffery v systému jsou drženy někde ve vyrovnávací paměti bufferů včetně nových, nepoužitých bloků. Tuto paměť sdílejí všechna fyzická bloková zařízení. V kterémkoliv okamžiku je ve vyrovnávací paměti řada blokových bufferů často v různých stavech, které patří vždy nějakému fyzickému zařízení. Každý blokový buffer použitý ke čtení dat z blokového zařízení nebo k jejich zápisu se ukládá ve vyrovnávací paměti bufferů. Časem může být z vyrovnávací paměti odstraněn aby se uvolnilo místo pro potřebnější buffer, nebo, pokud je používán často, může v paměti zůstávat dlouhou dobu.

Blokové buffery ve vyrovnávací paměti jsou jednoznačně identifikovány identifikátorem zařízení a číslem bloku, který je v bufferu uložen. Vyrovnávací paměť bufferů se skládá ze dvou částí. První část je seznam volných blokových bufferů. Existuje vždy jeden seznam pro jednu podporovanou velikost bufferů a volné blokové buffery v systému se v okamžiku svého vytvoření nebo vyprázdnění zařadí do příslušné fronty volných bufferů. Momentálně jsou podporovány buffery o velikosti 512, 1 024, 2 048, 4 096 a 8 192 bajtů. Druhou částí je samotná vyrovnávací paměť. Je to hashovací tabulka, která slouží jako vektor ukazatelů na řetězce bufferů se stejným hashovacím indexem. Hashovací index se generuje podle identifikátoru zařízení a čísla bloku. Na obrázku 9.7 vidíme hashovací tabulku a několik položek vyrovnávací paměti. Blokový buffer je buď v jednom ze seznamu volných bufferů, nebo ve vyrovnávací paměti.



Obrázek 9.7
Vyrovnávací paměť bufferů

Když je umístěn ve vyrovnávací paměti, je také zařazen v LRU seznamech. Pro každý typ bufferu existuje jeden LRU seznam, který systému umožňuje snadno realizovat potřebné operace nad daným typem bufferů, například zápis bufferů s novými daty na disk. Typ bufferu odráží jeho stav. Linux v současné době podporuje následující typy:

clean Nepoužité, nové buffery.

locked Buffer je uzamčen, čeká na zapsání.

dirty Buffer je modifikován. Obsahuje nová platná data a bude zapsán na disk, zatím ale zápis ještě nebyl naplánován.

shared Sdílené buffery.

unshared Buffery, které byly sdíleny, nyní však už sdíleny nejsou.

Vždy, když systém potřebuje z fyzického zařízení načíst buffer, zkouší jej nejprve najít ve vyrovnávací paměti bufferů. Pokud v ní buffer nenajde, vezme buffer potřebné délky ze seznamu volných bufferů a přemístí jej do vyrovnávací paměti. Pokud je buffer v paměti, může ale nemusí být aktuální. Pokud není aktuální nebo pokud je to nový blokový buffer, musí systém požádat ovladač zařízení o načtení příslušného bloku dat z disku.

Stejně jako všechny ostatní vyrovnávací paměti, i vyrovnávací paměť bufferů musí být spravována tak, aby pracovala efektivně a aby zajišťovala spravedlivou alokaci položek paměti mezi všemi blokovými zařízeními. K provádění řady potřebných úklidových operací ve vyrovnávací paměti používá Linux démona `bdflush`, některé operace nicméně probíhají automaticky jako přímý důsledek používání paměti.

9.3.1 Démon bdflush

Démon `bdflush` je démon jádra, který zajišťuje dynamickou reakci v případě, kdy je v systému příliš mnoho modifikovaných bufferů, tedy bufferů obsahujících data, která musejí být časem zapsána na disk. Spouští se jako vlákno jádra při startu systému a sám sebe nazývá „kflushd“, což je jméno, které uvidíte, když si příkazem `ps` zobrazíte procesy v systému. Většinu času démon spí a čeká, až počet modifikovaných bufferů dosáhne určité hranice. Když se alokují a ruší buffery, kontroluje se počet modifikovaných bufferů. Pokud dosáhne počet modifikovaných bufferů určitého procenta ze všech bufferů v systému, probouzí se démon `bdflush`. Implicitní hranice je 60 %, pokud je ale v systému nedostatek bufferů, bude démon probuzen dříve. Nastavená hodnota se dá zobrazit a měnit příkazem `update`:

```
# update -d

bdflush version 1.4
0: 60 Max fraction of LRU list to examine for dirty blocks
1: 500 Max number of dirty blocks to write each time bdflush activated
2: 64 Num of clean buffers to be loaded onto free list by refill_freelist
3: 256 Dirty block threshold for activating bdflush in refill_freelist
4: 15 Percentage of cache to scan for free clusters
5: 3000 Time for data buffers to age before flushing
6: 500 Time for non-data (dir, bitmap, etc) buffers to age before flushing
7: 1884 Time buffer cache load average constant
8: 2 LAV ratio (used to determine threshold for buffer fratricide).
```

Všechny modifikované buffery jsou umístěny v LRU seznamu `BUF_DIRTY` a démon `bdflush` se snaží vždy nějaký rozumný počet zapsat na disk. I tuto hodnotu je možno zobrazit a nastavit příkazem `update` (viz výše).

9.3.2 Proces update

Proces `update` není jen příkaz, je to zároveň i démon. Běží jako superuživatelský (v době inicializace systému) a periodicky vyprazdňuje všechny staré modifikované buffery na disk. Dosahuje toho voláním systémových rutin, které dělají více méně to samé, co démon `bdflush`. Vždy, když se dokončí modifikace bufferu, přidělí se mu čas, kdy má být zapsán na disk. Vždy, když `update` běží, prozkoumá všechny modifikované buffery v systému a hledá ty, které už mají být zapsány. Všechny takové buffery se zapíší na disk.

9.4 Souborový systém /proc

Souborový systém `/proc` ukazuje skutečnou sílu virtuálního souborového systému Linuxu. Fyzicky ve skutečnosti neexistují (zase jeden kouzelnický trik Linuxu) ani adresář `/proc`, ani jeho podadresáře a soubory. Jak tedy můžete pořídit výpis souboru `/proc/devices`? Souborový systém `/proc` se, stejně jako reálné souborové systémy, registruje ve virtuálním souborovém systému. Když však VFS tento systém volá a požaduje jeho inody při otevírání jeho souborů a adresářů, vytváří systém `/proc` tyto soubory a adresáře podle informací v jádře. Například soubor `/proc/devices` je generován z datových struktur jádra, popisujících jeho zařízení.

Souborový systém `/proc` poskytuje uživateli okno do interní činnosti jádra. V souborovém systému `/proc` vytváří své položky několik subsystémů Linuxu, například moduly jádra popsané v kapitole „Moduly“.

9.5 Speciální soubory zařízení.

Linux, stejně jako všechny verze systému Unix, představuje svá hardwarová zařízení jako soubory. Například `/dev/null` je nulové zařízení. Soubory zařízení nezabírají v souborovém systému žádný datový prostor, představují pouze přístupový bod k ovladači zařízení. Souborový systém `ext2` i virtuální souborový systém implementují soubory zařízení jako speciální typy inodů. Existují dva typy souborů zařízení: znakové a blokové soubory. I v samotném jádře implementuje ovladač zařízení souborové operace: můžete jej otevírat, zavírat a podobně. Znaková zařízení umožňují vstupně/výstupní operace ve znakovém režimu, bloková zařízení vyžadují veškerý přístup prostřednictvím vyrovnávací paměti bufferů. Velmi často pro některé subsystémy neexistují ovladače reálných zařízení, ale takzvané ovladače pseudozařízení, napří-

klad vrstva ovladače zařízení SCSI. Na soubory zařízení se odkazuje pomocí hlavního čísla, které identifikuje typ zařízení, a pomocí vedlejšího čísla, které identifikuje jednotku, instanci daného hlavního typu. Například disky IDE primárního řadiče IDE mají hlavní číslo 3 a první oblast každého disku IDE má vedlejší číslo 1. Tak nám výpis `ls -l` souboru `/dev/hda1` dává: **24**

```
$ brw-rw---- 1 root    disk       3,    1  Nov 24  15:09 /dev/hda1
```

V jádře je každé zařízení jednoznačně popsáno datovým typem `kdev_t`, který je dlouhý dva bajty, první bajt obsahuje vedlejší číslo zařízení, druhý obsahuje hlavní číslo zařízení. **25**

Výše uvedené zařízení IDE je v jádře označeno hodnotou *0x0301*. Inode systému ext2 reprezentující blokové nebo znakové zařízení má v ukazateli na první datový blok uloženo hlavní a vedlejší číslo zařízení. Když VFS takovýto inode načte, nastaví se identifikátor zařízení do položky `i_rdev` inodu VFS.

Odkazy na zdrojové texty jádra

1 – Viz `fs/ext2/*`
2 – Viz `include/linux/-ext2_fs_i.h`
3 – Viz `include/linux/-ext2-fs_sb.h`
4 – Viz `ext2_group_desc in include/-linux/ext2:fs.h`
5 – Viz `ext2_dir_entry in include/-linux/ext2_fs.h`
6 – Viz `ext2_new_block() in fs/ext2/-balloc.c`
7 – Viz `fs/*`
8 – Viz `fs/inode.c`
9 – Viz `fs/buffer.c`
10 – Viz `fs/dcache.c`
11 – Viz `include/-linux/fs.h`
12 – Viz `include/-linux/fs.h`
13 – Viz `sys_setup() in fs/-filesystems.c`
14 – Viz `file_system_type in include/-linux/fs.h`
15 – Viz `do_mount() in fs/super.c`
16 – Viz `get_fs_type() in fs/super.c`
17 – Viz `add_vfsmnt() in fs/super.c`
18 – Viz `do_umount() in fs/super.c`
19 – Viz `remove_vfsmnt() in fs/super.c`
20 – Viz `fs/inode.c`

21 – `Viz fs/dcache.c`

22 – `Viz bdflush() in fs/buffer.c`

23 – `Viz sys bdflush() in fs/buffer.c`

24 – `Viz /include/linux/major.h for all of Linux's major device numbers`

25 – `Viz include/-linux/kdev_t.h`

Kapitola

10

Sítě

Termíny sítě a Linux jsou prakticky synonyma. Linux je fakticky produktem Internetu nebo WWW. Jeho tvůrci a uživatelé používají Internet k výměně nápadů a samotný Linux se často používá k zajištění síťových potřeb různých organizací. V této kapitole se popisuje jak Linux podporuje síťové protokoly souhrnně označované jako TPC/IP.

Protokoly TCP/IP byly navrženy pro komunikaci počítačů připojených k síti ARPANET, americké výzkumné sítě financované vládou USA. ARPANET vedl ke vzniku základních technik počítačových sítí, jako jsou přepínání paketů a vrstvení protokolů, kdy jeden protokol poskytuje služby dalšímu. ARPANET byl oficiálně zrušen v roce 1988, jeho následníci (NSFNET[1] a Internet) se však stále rozrůstají. To co dnes označujeme jako World Wide Web je vybudováno právě na ARPANETu a protokolech TCP/IP. Unix byl na ARPANETu velmi používán a jeho první verze s podporou sítí byla verze BSD 4.3. Implementace síťových služeb v Linuxu je postavena na BSD verzi 4.3 v tom, že (s určitými výjimkami) podporuje BSD sokety a plně implementuje podporu TCP/IP. Toto programové rozhraní bylo zvoleno jednak kvůli jeho oblibě a jednak aby se napomohlo přenositelnosti aplikací mezi Linuxem a jinými platformami Unixu.

10.1 Přehled TCP/IP

V této části popisujeme základní principy sítí na bázi TCP/IP. Nejedná se o vyčerpávající přehled, doporučuji vám ale přečíst si jej. V IP síti má každý počítač přiřazenu IP adresu, což je 32bitové číslo jednoznačně popisující daný počítač. Web je velmi rozsáhlý a stále roste. Každá IP síť a každý k ní připojený počítač musí mít přiřazenu jedinečnou IP adresu. IP adresy

1 National Science Foundation Network

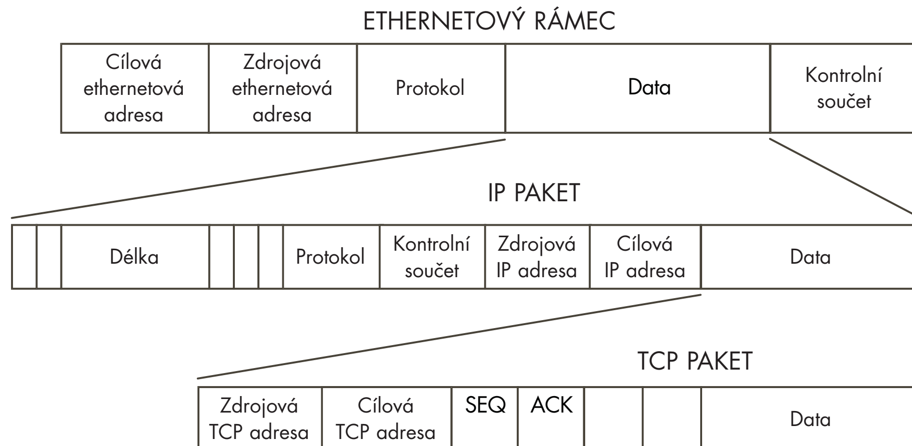
se reprezentují jako čtyři čísla oddělená tečkami, například `16.42.0.9`. IP adresa se skládá ze dvou částí, z *adresy sítě* a *adresy počítače*. Velikosti těchto částí mohou být různé (existuje několik tříd IP adres), pokud ale jako příklad vezmeme adresu `16.42.0.9`, pak adresa sítě je `16.42` a adresa počítače `0.9`. Adresa počítače se dále může dělit na *podsíť* a adresu počítače. Pokud znovu použijeme jako příklad adresu `16.42.0.9`, pak podsíť může být `16.42.0` a na ní je počítač `9`. Toto dělení IP adres umožňuje organizacím strukturovat své sítě. Například `16.42` může být adresa sítě společnosti ACME Computer Company, `16.42.0` bude jejich podsíť `0`, `16.42.1` podsíť `1`. Tyto podsítě mohou být v různých budovách, propojených pevnou linkou nebo radiovým spojem. IP adresy přiřazuje administrátor sítě a použití podsítí je dobrá metoda, jak administraci distribuovat. Administrátor podsítě pak může libovolně přiřazovat adresy v rámci své podsítě.

Obecně vzato se IP adresy špatně pamatují, daleko příjemnější jsou jména. Jméno `linux.acme.com` se pamatuje daleko snáze než `16.42.0.9`, potřebujeme ovšem nějaký mechanismus pro konverzi síťových jmen na IP adresy.

Jména mohou být staticky uvedena v souboru `/etc/hosts`, nebo Linux může o překlad jména na adresu požádat server DNS. V takovém případě musí lokální počítač znát adresu jednoho nebo více serverů DNS, které se udávají v souboru `/etc/resolv.conf`.

Vždy, když se připojíte k jinému počítači, řekněme při prohlížení webovské stránky, komunikujete s ním prostřednictvím jeho IP adresy. Tato data jsou uložena v IP paketech, z nichž každý má hlavičku, která obsahuje IP adresy zdrojového a cílového počítače, kontrolní součet a další užitečné údaje. Kontrolní součet se odvozuje z dat v IP paketu a umožňuje příjemci paketu rozhodnout, zda paket nebyl při přenosu poškozen například rušením na telefonní lince. Data odesílaná aplikací se mohou dělit na menší pakety, s nimiž se snáze manipuluje. Velikost datových paketů závisí na použitém fyzickém síťovém médiu, například ethernetové pakety jsou obecně větší než pakety PPP. Cílový počítač musí zpětně sestavit pakety dohromady a poté je teprve předá přijímající aplikaci. Tuto fragmentaci a zpětné skládání dat můžete vidět graficky, když přistupujete k webovské stránce s řadou obrázků prostřednictvím poměrně pomalé sériové linky.

Počítače připojené ke stejné podsíti si mohou mezi sebou vyměňovat IP pakety přímo, všechny ostatní pakety budou odesílány na speciální počítač, *bránu*. Brána (nebo router) je připojena k více než jedné podsíti a předává pakety zachycené na jedné podsíti patřící na jinou. Pokud budeme mít například podsítě `16.42.1.0` a `16.42.0.0` propojeny branou, musí se všechny pakety z podsítě `0` určené na podsíť `1` předávat bráně, která zajistí jejich směrování. Lokální počítač si sestavuje směrovací tabulku, která mu umožňuje směrovat IP pakety na správné brány. Pro každou cílovou IP adresu obsahuje směrovací tabulka údaje, které Linuxu říkají, kterému počítači má paket poslat, aby se dostal na místo určení. Tyto směrovací tabulky jsou dynamické a mění se podle toho, jak aplikace používají síť a jak se mění topologie sítě.



Obrázek 10.1
Protokolové vrstvy TCP/IP

Protokol IP je protokol transportní vrstvy, který ostatním protokolům zajišťuje přenos jejich dat. Protokol TCP (Transmission Control Protocol) je protokol zajišťující spolehlivé dvoubodové spojení, který pro příjem a vysílaní svých paketů používá protokol IP. Tak jako má svou hlavičku IP paket, má svou hlavičku i TCP paket. TCP je protokol s navazováním spojení, kdy jsou dvě síťové aplikace spojeny jedním virtuálním spojem, přestože ten může vést přes řadu podsítí, bran a routerů. TCP spolehlivě přijímá a odesílá data mezi těmito dvěma aplikacemi a zaručuje, že nedojde k žádné ztrátě ani zdvojení paketů. Když TCP odesílá svůj paket prostřednictvím IP, data obsažená v IP paketu jsou celý TCP paket. IP vrstvy obou komunikujících počítačů zodpovídají za odesílání a příjem IP paketů. Protokol UDP (User Datagram Protocol) rovněž používá jako transportní protokol IP, na rozdíl od TCP však UDP není spolehlivý protokol a zajišťuje datagramové služby. Toto využití protokolu IP jako nosiče jiných protokolů předpokládá, že přijímající IP vrstva musí vědět, jaké vyšší protokolové vrstvě musí předat data IP paketu. Proto je v hlavičce každého IP paketu hodnota, obsahující identifikátor protokolu. Když TCP požádá IP vrstvu o přenos TCP paketu, bude hlavička IP paketu obsahovat údaj o tom, že se přenáší TCP paket. Přijímající IP vrstva se podle tohoto identifikátoru rozhoduje, jaké vrstvě data předat, v tomto případě je předá TCP vrstvě. Když aplikace komunikují pomocí TCP/IP, musejí udávat nejen cílovou IP adresu, ale také adresu *portu* aplikace. Adresa portu jednoznačně identifikuje aplikaci a standardní síťové aplikace používají standardní porty, například webový server používá port 80. Registrované porty je možno vidět v souboru `/etc/services`.

Vrstvení protokolů nekončí u TCP, UDP a IP. Samotná vrstva IP používá řadu různých fyzických médií k přenosu IP paketů na jiné počítače. Tato média sama mohou přidávat vlastní protokolové hlavičky. Jedním příkladem může být ethernetová vrstva, vrstvy PPP a SLIP fungu-

jí zase jinak. Ethernetová síť umožňuje propojit současně řadu počítačů fyzicky jedním kabelem. Každý vysílaný etherentový rámec se tak dostane ke všem připojeným počítačům, a proto má každé ethernetové zařízení jedinečnou adresu. Každý ethernetový rámec vyslaný na nějakou adresu bude přijat adresovaným počítačem a všechny ostatní připojené počítače jej budou ignorovat. Tato jedinečná adresa je nastavena každému ethernetovému zařízení při výrobě a obvykle je uložena v paměti SROM[2] na ethernetové kartě. Ethernetové adresy mají délku 6 bajtů, například `08-00-2b-00-49-A4`. Některé ethernetové adresy jsou rezervovány pro potřeby hromadného vysílání a rámce poslané na takovéto adresy budou zachyceny všemi připojenými počítači. Protože ethernetový frame může (jako data) nést řadu rozdílných protokolů, podobně jako IP paket obsahuje v hlavičce identifikátor protokolu. Díky tomu může ethernetová vrstva správně postoupit přijaté rámce IP vrstvě.

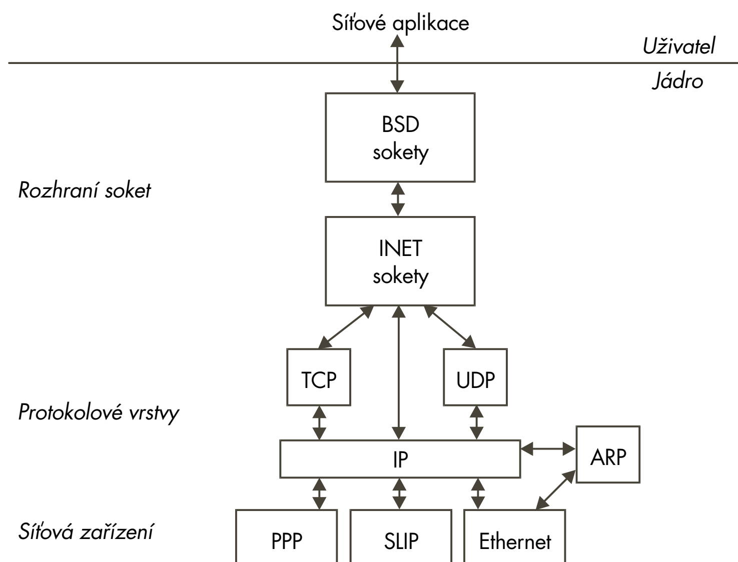
Aby mohla IP vrstva poslat IP paket protokolem jako je ethernet, musí zjistit ethernetovou adresu cílového počítače. IP adresy představují totiž pouze adresační koncept, ethernetová zařízení mají své vlastní fyzické adresy. IP adresy může administrátor sítě přiřazovat a měnit podle potřeb, ovšem síťový hardware reaguje pouze na svou vlastní ethernetovou adresu nebo na speciální hromadně vysílané pakety, které přijímají všechna zařízení. Linux používá k překladu IP adres na reálné hardwarové adresy, jako je například ethernetová adresa, protokol ARP (Address Resolution Protocol). Počítač, který potřebuje zjistit hardwarovou adresu odpovídající určité IP adrese, pošle požadavek ARP obsahující IP adresu, která se má přeložit hromadným vysílacím mechanismem, takže tento paket zachytí všechny počítače na sítí. Počítač, kterému patří požadovaná IP adresa, odpoví paketem ARP, v němž je uvedena jeho fyzická adresa. ARP není omezen pouze na ethernetová zařízení, dokáže zajistit překlad IP adres i na jiných médiích, například na sítích FDDI. Zařízení, která neumějí reagovat na protokol ARP, jsou označena tak, že se s nimi Linux nepokouší tímto protokolem komunikovat. Existuje také obrácená funkce, reverzní ARP nebo RARP, která překládá fyzické hardwarové adresy na IP adresy. Tuto funkci používají brány, které reagují na žádosti ARP z IP adres, které jsou na vzdálené síti.

10.2 Síťové vrstvy TCP/IP v Linuxu

Stejně jako samotné síťové protokoly, i Linux implementuje internetovou rodinu protokolů pomocí několika vzájemně propojených vrstev softwaru, jak můžeme vidět na obrázku 10.2. Sokety BSD poskytují programům služby obecné správy soketů. Tato vrstva je podporována soketovou vrstvou INET, která řídí komunikaci koncových bodů protokolů TCP a UDP. UDP (User Datagram Protocol) je protokol bez navazování spojení, zatímco TCP (Transmission Control Protocol) je spolehlivý dvoubodový protokol. Když se vysílají UDP pakety, Linux neví a nestará se, zda pakety správně dojdou k cíli. TCP pakety jsou číslovány a oba konce spo-

2 Synchronous Read Only Memory

ení TCP zajišťují, že vysílaná data budou správně přijata. IP vrstva obsahuje kód implementující IP protokol. Tento kód připojuje k vysílaným datům IP hlavičky a ví, jak má příchozí IP pakety předávat vrstvám TCP nebo UDP. Pod IP vrstvou, která slouží jako podpora veškerého síťového software v Linuxu, jsou síťová zařízení, například PPP a ethernet. Síťová zařízení nemusejí vždy reprezentovat nějaké fyzické zařízení. Některá, například zpětné zařízení, jsou záležitosti čistě softwarové. Na rozdíl od standardních zařízení v Linuxu, která se vytvářejí příkazem `mknod`, se síťová zařízení objevují pouze v případě, že je příslušný síťový software nalezne a zinicializuje. Zařízení `eth0` uvidíte pouze v případě, že jádro má vestavěn příslušný ovladač ethernetového zařízení. Protokol ARP je umístěn mezi IP vrstvou a protokoly, které podporují adresaci pomocí ARP.



Obrázek 10.2
Síťové vrstvy v Linuxu

10.3 Soketové rozhraní BSD

Jedná se o obecné rozhraní, které podporuje nejenom různé formy síťové komunikace, ale slouží také jako meziprocesový komunikační mechanismus. Soket popisuje jeden konec komunikační linky, dva komunikující procesy budou mít každý svůj soket, popisující jejich konce společné komunikační linky. Soket je možno chápat jako zvláštní případ roury, na rozdíl od rour, ale nemá soket žádné omezení objemu dat, která může obsahovat. Linux podporuje několik tříd soketů, které jsou známé jako *adresové rodiny*. Je to dáno tím, že každá třída má své vlastní adresační metody. Linux podporuje následující soketové adresové rodiny či domény:

UNIX	Sokety unixové domény
INET	Adresová rodina Internetu podporuje komunikaci protokoly TCP/IP
AX25	Amateur radio X25
IPX	Novell IPX
APPLETALK	Appletalk DDP
X25	X25

Existuje několik typů soketů, které reprezentují typ služby, podporující spojení. Ne všechny adresové rodiny podporují všechny typy služeb. Sokety BSD podporují řadu soketových typů:

stream Tyto sokety poskytují spolehlivé obousměrné sekvenční datové proudy se zajištěním, že při přenosu nemůže dojít ke ztrátě, porušení nebo duplikaci dat. Streamy jsou podporovány TCP protokolem v adresové rodině INET.

datagram Tyto sokety rovněž zajišťují obousměrný datový přenos, na rozdíl od streamů však nezaručují doručení zpráv. I když zpráva dorazí, není zaručeno, že dorazí ve správném pořadí nebo neduplikovaně a neporušeně. Tento typ soketů je podporován UDP protokolem z adresové rodiny INET.

raw Umožňuje procesům přímý přístup k nízkoúrovňovým protokolům. Je například možné otevřít raw soket na ethernetové zařízení nebo sledovat nízkoúrovňový IP datový provoz.

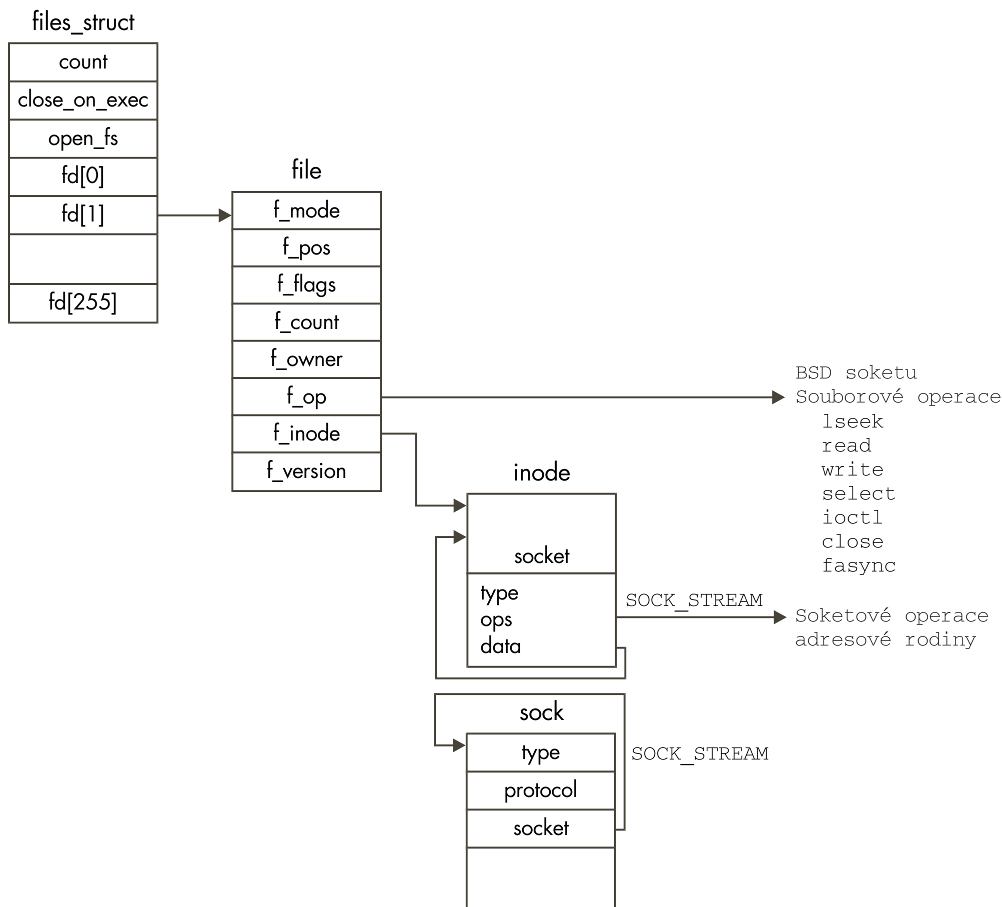
reliable delivered messages
Velmi se podobají datagramům, je však zajištěno doručení dat.

sequenced packet Velmi se podobají streamům, velikost datových paketů je ale pevná.

packet Toto není standardní BSD soketový typ, jedná se o rozšíření Linuxu, které umožňuje procesům přistupovat přímo k paketům na úrovni zařízení.

Procesy komunikující prostřednictvím soketů používají model klient/server. Server nabízí službu a klient tuto službu využívá. Jedním příkladem může být webový server, který nabízí webovské stránky, a webový klient, prohlížeč, který tyto stránky čte. Server, který používá sokety, musí soket nejprve vytvořit a poté mu přidělit jméno. Formát jména závisí na adresové rodině soketu a ve svém důsledku představuje jméno lokální adresu serveru. Jméno nebo adresa soketu se udává pomocí datové struktury `sockaddr`. Soket rodiny INET bude mít přiřazenu IP adresu. Čísla registrovaných portů můžeme zjistit v souboru `/etc/services`, například webový server používá port 80. Po přiřazení adresy soketu pak server naslouchá příchozím spojením a čeká na požadavky na přidělenou adresu. Původce požadavku, klient, vy-

tvoří soket a předává jím žádost o spojení, přičemž jako cílovou adresu udává adresu serveru. Pro soket INET je adresou serveru jeho IP adresa a číslo portu. Žádost musí projít různými protokolovými vrstvami a nakonec skončí v soketu, na němž server poslouchá. Když server žádost přijme, může ji buď akceptovat, nebo odmítnout. Pokud ji akceptuje, musí server vytvořit nový soket, na němž žádost zpracuje. Soket používaný k příjmu požadavků se totiž nedá zároveň použít k jejich plnění. Po navázání spojení mohou obě strany volně posílat data. Po ukončení přenosů přestává být spojení zapotřebí a může se zrušit. Po celou dobu se zajišťuje, aby bylo s datovými pakety správně naloženo.



Obrázek 10.3
Datové struktury BSD soketu

Přesný význam operací nad BSD sokety záleží na adresové rodině, kde jsou sokety implementovány. Vytvoření TCP/IP spojení je velmi odlišné od navázání spojení pomocí X.25. Stejně jako virtuální souborový systém, i BSD soketovou vrstvu používá Linux jako abstrakci od konkrétních síťových mechanismů. Aplikace pracuje s rozhraním BSD vrstvy, která zajišťuje provedení všech operací příslušnými programy příslušných adresových rodin. V době inicializace jádra se adresové rodiny vestavěné v jádře registrují u BSD rozhraní. Později, když aplikace vytváří a používá BSD sokety, se vytvoří asociace mezi BSD soketem a příslušnou adresovou rodinou. Tyto asociace se vytvářejí pomocí křížových datových struktur a tabulek adres rutin, kterými adresové rodiny nabízejí své specifické služby. Existuje například pro adresovou rodinu specifická rutina pro vytvoření soketu, kterou BSD rozhraní používá, když aplikace chce vytvořit soket.

Při konfiguraci jádra se do vektoru `protocols` vkládá řada adresových rodin a protokolů. Každá položka je reprezentována svým jménem, například „INET“, a adresou inicializační rutiny. Při inicializaci soketového rozhraní v době zavádění systému se postupně volají jednotlivé inicializační rutiny. U adresových rodin vede toto volání k registraci sady protokolových operací. Jedná se o sadu rutin, z nichž každá provádí určitou operaci specificky pro danou adresovou rodinu. Registrované protokolové operace se ukládají ve vektoru `pops`, vektoru ukazatelů na datové struktury `proto_ops`.

1

Datová struktura `proto_ops` se skládá z typu adresové rodiny a z ukazatelů na rutiny soketových operací, specifické pro danou adresovou rodinu. Vektor `pops` je indexován identifikátorem adresové rodiny, například rodiny INET (AF_INET je 2).

10.4 Soketová vrstva INET

Soketová vrstva INET podporuje internetovou adresovou rodinu, která obsahuje protokoly TCP/IP. Jak už bylo popsáno dříve, tyto protokoly jsou vrstveny, jeden protokol využívá služby jiného. TCP/IP kód a datové struktury v Linuxu odpovídají tomuto vrstvení. Prostřednictvím skupiny soketových operací internetové rodiny, které se registrují v době inicializace sítě, se poskytuje rozhraní soketové vrstvě BSD. Soketová vrstva BSD pak volá podpůrné rutiny internetové vrstvy z datové struktury `proto_ops` této registrované vrstvy. Například žádost o vytvoření BSD soketu s uvedením adresy ve vrstvě INET povede k volání rutiny pro vytvoření soketu v internetové vrstvě. U všech těchto operací předává soketová vrstva BSD internetové vrstvě datovou strukturu `socket` reprezentující BSD soket. Aby nevznikaly zmatky tím, že se struktura `socket` naplní informacemi specifickými pro protokoly TCP/IP, používá internetová vrstva vlastní strukturu `sock`, kterou propojuje se strukturou `socket` BSD vrstvy. Toto propojení je znázorněno na obrázku 10.3. Datová struktura `sock` je propojena s datovou strukturou `socket` pomocí ukazatele `data` v BSD soketu. Soketová volání do vrstvy INET tak mohou snadno získat datovou strukturu `sock`. Při vytvoření datové

2

struktury sock se nastaví také ukazatel protokolových operací, který závisí na používaném protokolu. Pokud byl požadován protokol TCP, bude ukazatel protokolových operací ukazovat na sadu TCP operací, potřebných pro práci s TCP spojením.

10.4.1 Vytvoření BSD soketu

Systémové volání pro vytvoření nového soketu přebírá identifikátory adresové rodiny, typu soketu a protokolu.

Podle požadované adresové rodiny se nejprve ve vektoru pops nalezne požadovaná rodina. Může se stát, že určitá adresová rodina je implementována jako modul jádra a v takovém případě bude muset démon kerneld příslušný modul nejprve nahrát. Dále se alokuje nová datová struktura socket reprezentující vytvářený BSD soket. Datová struktura socket je ve skutečnosti fyzicky součástí datové struktury inode VFS a alokace soketu fakticky představuje alokaci inodu VFS. Může to vypadat na první pohled podivně, je ale třeba vzít v úvahu, že se sokety se pracuje stejným způsobem jako s běžnými soubory. Tak jak jsou všechny soubory reprezentovány strukturou VFS inode kvůli zajištění souborových operací, i BSD sokety je nutné ze stejných důvodů reprezentovat jako inody VFS.

Nově vytvořená datová struktura socket obsahuje ukazatel na specifické soketové operace požadované adresové rodiny, který ukazuje na datovou strukturu proto_ops zjištěnou z vektoru pops. Typ se nastaví podle požadovaného typu soketu na jednu z hodnot SOCK_STREAM, SOCK_DGRAM a podobně. Pak se volá rutina vytvoření soketu specifická pro danou adresovou rodinu, jejíž adresa je uložena v datové struktuře proto_ops.

Z vektoru fd aktuálního procesu se alokuje volný deskriptor souboru a inicializuje se datová struktura file, na níž deskriptor ukazuje. Tato inicializace zahrnuje nastavení ukazatele souborových operací na souborové operace BSD soketu, které jsou zajišťovány BSD soketovým rozhraním. Všechny operace se dále směrují na soketové rozhraní, které je předává příslušné adresové rodině prostřednictvím volání rutin specifické adresové rodiny.

10.4.2 Přiřazení adresy BSD soketu INET

Aby mohl server poslouchat a čekat na příchozí žádosti o spojení, musí si vytvořit soket INET BSD a přiřadit mu adresu. Operace přiřazení je z větší části obsluhována soketovou vrstvou INET s částečnou podporou nižších protokolových vrstev TCP a UDP. Soket s přiřazenou adresou se nedá použít k žádné jiné komunikaci. Znamená to, že status ve struktuře socket musí být TCP_CLOSE. Adresa sockaddr předávaná operaci přiřazení obsahuje IP adresu a nepovinně číslo portu. Obvykle se přiřazuje IP adresa přidělená některému síťovému zařízení, které podporuje adresovou rodinu INET, je aktivní a dá se použít. Která síťová rozhraní jsou v systému momentálně aktivní, můžete zjistit příkazem ifconfig. Může se také

přiřadit vysílací IP adresa se samými nulami nebo samými jedničkami. Jedná se o speciální adresy s významem „poslat všem“. IP adresa může být také volena libovolně pokud počítač pracuje jako transparentní proxy-server nebo firewall, přiřazení adresy ovšem může provést pouze proces s oprávněními superuživatele. IP adresa přiřazená soketu se ukládá v datové struktuře `sock` v položkách `recv_addr` a `saddr`. Používají se při hashovacím vyhledávání a jako adresa odesílatele. Číslo portu je nepovinné a pokud není uvedeno, požádá se příslušná podpůrná síťová vrstva o přiřazení volného portu. Konvencí je dáno, že porty s čísly menšími než 1 024 mohou používat pouze superuživatelské procesy. Pokud se port přiřazuje automaticky podpůrnou síťovou vrstvou, je vždy přiřazeno číslo větší než 1 024.

Paket přijatý nějakým síťovým zařízením se musí předat správným soketům INET a BSD, které jej zpracují. Z toho důvodu udržují UDP a TCP hashovací tabulky, které se používají k nalezení adresy při zachycení IP paketu a jeho předávání správné dvojici `socket`/`sock`. TCP je protokol s navazováním spojení, takže zpracování TCP paketu je podstatně složitější než zpracování UDP paketu.

UDP udržuje hashovací tabulku alokovaných UDP portů, tabulku `udp_hash`. Skládá se z ukazatelů na datové struktury `sock` a indexuje se hashovací funkcí založenou na čísle portu. Protože hashovací tabulka je podstatně menší než rozsah dostupných čísel portů (`udp_hash` má pouze 128 položek, respektive `UDP_HTABLE_SIZE` položek), některé položky z tabulky ukazují na řetězce datových struktur `sock`, které jsou propojeny pomocí ukazatele `next` ve struktuře `sock`.

TCP je podstatně složitější a udržuje několik hashovacích tabulek. TCP ovšem nepřidává datovou strukturu `sock` při přiřazování adresy, v té době pouze kontroluje, zda požadovaný port už nebyl přidělen. Datová struktura `sock` se přidává do hashovacích tabulek TCP až při vyvolání operace *listen*.

10.4.3 Vytvoření spojení na BSD soketu

Jakmile soket vytvoříme a nepoužíváme jej k poslouchání příchozích žádostí o spojení, můžeme jej použít k vytvoření odchozí žádosti o navázání spojení. U protokolů bez navazování spojení, jako je například UDP, tato operace nic moc neobnáší, ovšem u protokolů s navazováním spojení (jako je TCP) to obnáší vytvoření virtuálního spoje mezi dvěma aplikacemi.

Odchozí spojení je možno navázat pouze na BSD soket INET, který je ve vhodném stavu - tedy takový, který nemá doposud žádné spojení navázáno a nepoužívá se k zachycování příchozích požadavků. Znamená to, že status ve struktuře `socket` musí být `SS_UNCONNECTED`. UDP protokol nenavazuje virtuální spoj mezi dvěma aplikacemi, všechny jím odesílané zprávy jsou datagramy, tedy zprávy, které mohou, ale nemusejí dorazit ke svému cíli. I tento protokol ale podporuje BSD soketovou operaci *connect*. Operace navázání spojení na UDP soketu jednodu-

še nastaví adresy vzdálené aplikace, tedy její IP adresu a číslo IP portu. Dále se nastavuje vyrovnávací paměť položek směrovací tabulky, takže UDP pakety posílané na daný soket nebudou muset opakovaně načítat směrovací databázi (dokud původní trasa zůstane platná). Na směrovací informace ve vyrovnávací paměti ukazuje položka ip_route_cache v datové struktuře sock. Pokud nejsou zadány žádné adresovací informace, použijí se při odesílání zpráv daných BSD soketem právě směrovací informace a IP adresační informace uložené ve vyrovnávací paměti. Nakonec převede UDP strukturu sock do stavu TCP_ESTABLISHED.

Operace navázání spojení na TCP soketu musí nejprve vytvořit TCP zprávu s informacemi o požadavku na spojení a odeslat ji na zadanou cílovou IP adresu. TCP zpráva obsahuje informace o spojení, počáteční sekvenční číslo, maximální velikost zprávy, kterou je schopen žadatel zpracovat, velikost příjmového a odesílacího okna a další. V rámci protokolu TCP jsou všechny zprávy číslovány a počáteční sekvenční číslo se použije jako číslo první zprávy. Linux volí rozumně náhodné hodnoty, aby se předešlo některým typům protokolových útoků. Každá zpráva odeslaná jedním koncem TCP spojení a úspěšně přijatá druhým koncem spojení je potvrzena, takže odesílatel ví, že zpráva dorazila v pořádku a nepoškozená. Nepotvrzené zprávy se posílají znovu. Velikost vysílacího a příjmového okna udává počet zpráv, které je možno odeslat najednou bez potvrzení. Maximální velikost zprávy vychází ze síťového zařízení používaného iniciátorem spojení. Pokud druhá strana spojení podporuje jinou maximální velikost zprávy, použije se vždy minimum z obou hodnot. Aplikace navazující TCP spojení musí počkat na odpověď od vzdálené aplikace, která požadavek buď přijme, nebo odmítne. Protože TCP sock nyní čeká na příchozí zprávu, zařadí se do struktury tcp_listening_hash, která zajišťuje předávání došlých TCP zpráv správné struktuře sock. TCP rovněž spustí časovač, takže pokud na požadavek nepřijde do nějaké doby odpověď, požadavek propadne.

10.4.4 Poslouchání na BSD soketu INET

Když má soket přiřazenu adresu, může poslouchat a čekat na příchozí požadavky na spojení na přiřazené adrese. Síťová aplikace může poslouchat soketem bez předchozího přiřazení adresy, v takovém případě soketová vrstva INET přiřadí soketu automaticky nepoužité číslo portu (v daném protokolu). Soketová funkce poslechu převede soket do stavu TCP_LISTEN a provede všechny síťově závislé operace potřebné k příjmu žádostí.

U UDP soketů stačí pouze změnit stav soketu, TCP však přidává aktivovanou strukturu sock do dvou hashovacích tabulek. Jde o tabulky tcp_bound_hash a tcp_listening_hash. Obě se indexují prostřednictvím hashovací funkce založené na čísle IP portu.

Kdykoliv aktivní naslouchající soket přijme příchozí požadavek na navázání TCP spojení, vytvoří pro něj TCP novou datovou strukturu sock. Tato datová struktura sock se stává základem TCP spojení, pokud bude skutečně akceptováno. Dále se naklonuje příchozí

sk_buff obsahující žádost o spojení a zařadí se do fronty receive_queue naslouchající datové struktury sock. Klon bufferu sk_buff obsahuje ukazatel na nově vytvořenou datovou strukturu sock.

10.4.5 Příjem požadavku na spojení

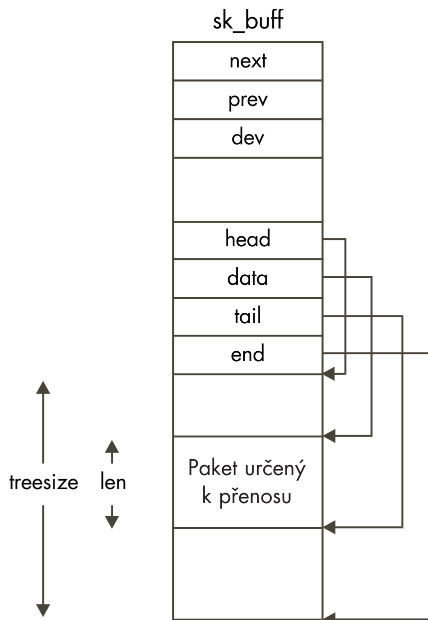
Protokol UDP nepodporuje koncept spojení, takže příjem požadavku na navázání spojení se týká pouze protokolu TCP, u něhož operace přijetí požadavku vede k vytvoření nové datové struktury socket z původního naslouchajícího soketu. Operace přijetí požadavku se pak předá příslušné podpůrné protokolové vrstvě, v tomto případě INET, aby se zajistilo přijetí příchozích požadavků. Protokolová vrstva INET nenechá operaci *accept* proběhnout, pokud podpůrný protokol, řekněme UDP, nepodporuje navazování spojení. V opačném případě se operace *accept* předá protokolu, řekněme TCP. Operace *accept* může být buď blokující, nebo neblokující. U neblokující operace končí operace neúspěšně, pokud není žádné příchozí spojení, které by mělo být přijato, a datová struktura socket se zruší. U blokující operace se síťová aplikace provádějící tuto operaci přidá do čekací fronty a je pozastavena, dokud nebude přijat TCP požadavek. Jakmile dojde požadavek, buffer sk_buff s žádostí se zruší a datová struktura sock je vrácena vrstvě INET, kde je přiřazena dříve vytvořené datové struktuře socket. Síťové aplikaci se vrací deskriptor souboru (fd) nového soketu a aplikace pak může tento deskriptor používat k soketovým operacím nad nově vytvořeným BSD soketem.

10.5 Vrstva IP

10.5.1 Soketové buffery

Jedním z problémů použití mnoha vrstev síťových protokolů, kdy jeden využívá služeb dalšího, spočívá v tom, že každý protokol potřebuje k odesílaným datům přidávat své hlavičky a patičky a při přijetí dat je musí naopak odstraňovat. Tím se komplikuje předávání datových bufferů mezi protokoly, protože každá vrstva musí zjišťovat, kde jsou její hlavičky a patičky. Jedním řešením by bylo kopírování části bufferu v každé vrstvě, to by ovšem bylo neefektivní. Namísto toho používá Linux k předávání dat mezi protokolovými vrstvami a ovladači síťových zařízení datovou strukturu sk_buff. Struktura sk_buff obsahuje ukazatele a délky, takže jednotlivé protokolové vrstvy mohou s daty aplikace pracovat pomocí standardních funkcí či metod.

4 Na obrázku 10.4 vidíme datovou strukturu sk_buff, ke každé takové struktuře je dále přiřazen blok dat. Struktura sk_buff má čtyři datové ukazatele, které slouží k manipulaci a úpravám nad daty soketového bufferu:



Obrázek 10.4
Soketový buffer (sk_buff)

head Ukazuje na začátek datové oblasti v paměti. Tato hodnota je dána v okamžiku, kdy se alokuje struktura sk_buff a její datová oblast.

data Ukazuje na momentální začátek protokolových dat. Tento ukazatel se mění podle toho, která protokolová vrstva momentálně strukturu sk_buff vlastní.

tail Ukazuje na momentální konec protokolových dat. Tato hodnota opět závisí na protokolu, který strukturu vlastní.

end Ukazuje na konec datové oblasti v paměti. Hodnota je pevně dána při alokaci bufferu.

Dále struktura obsahuje dvě délkové položky, len a truesize, které obsahují délku paketu v aktuálním protokolu a celkovou velikost datového bufferu. Obslužný kód struktury sk_buff poskytuje standardní mechanismy pro přidávání a rušení hlaviček a patiček jednotlivých protokolů. Tyto operace zajišťují koherentní manipulace s položkami data, tail a len:

push Přesune ukazatel data směrem k začátku datové oblasti a inkrementuje údaj len. Používá se při přidávání dat nebo protokolových hlaviček na začátek odesílaných dat.

6 **pull** Přesouvá ukazatel `data` dále od začátku, směrem ke konci datové oblasti, a dekrementuje údaj `len`. Používá se při rušení dat nebo protokolových hlaviček ze začátku oblasti.

7 **put** Přesouvá ukazatel `tail` směrem od začátku a inkrementuje údaj `len`. Používá se při přidávání dat nebo protokolových informací na konec stávajících dat.

8 **trim** Přesouvá ukazatel `tail` směrem k začátku oblasti a dekrementuje údaj `len`. Používá se při rušení dat nebo protokolových patiček z konce stávajících dat.

Datová struktura `sk_buff` dále obsahuje ukazatele používané pro ukládání struktur do obousměrně propojeného kruhového seznamu při jejich zpracovávání. Existují obecné funkce nad strukturou `sk_buff`, které zajišťují přidávání a rušení struktury na začátek a konec těchto seznamů.

10.5.2 Příjem IP paketů

V kapitole „Ovladače zařízení" je popsáno, jak se ovladače síťových zařízení přidávají do jádra a inicializují. Výsledkem je několik datových struktur `device` propojených v seznamu `dev_base`. Každá datová struktura `device` popisuje své zařízení a poskytuje množinu operací, které volají síťové protokoly, když potřebují, aby ovladač síťového zařízení provedl nějakou operaci. Tyto funkce se vesměs týkají odesílání dat a práce s adresou síťového zařízení. Když síťové zařízení zachytí paket ze sítě, musí přijatá data zkonvertovat na strukturu 9 `sk_buff`. Tyto struktury `sk_buff` se pak přidávají do fronty `backlog` síťových zařízení.

Když se fronta `backlog` příliš rozroste, přijaté buffery `sk_buff` se ruší. Nastaví se příznak spuštění síťového bottom-half ovladače, který bude muset zajistit zpracování přijatých paketů.

Když plánovač spustí síťový bottom-half ovladač, zpracují se nejprve všechny síťové pakety 10 čekající na odeslání a poté se zpracovávají buffery `sk_buff` ve frontě `backlog` a rozhoduje se, které protokolové vrstvě se mají přijaté pakety předat.

Při inicializaci síťového systému se každý protokol registruje přidáním datové struktury `packet_type` buď do seznamu `ptype_all`, nebo do hashovací tabulky `ptype_base`. Datová struktura `packet_type` obsahuje typ protokolu, ukazatel na síťové zařízení, ukazatel na rutinu zpracování přijatých dat a konečně ukazatel na další strukturu `packet_type` v seznamu nebo v hashovacích řetězcích. Seznam `ptype_all` může sloužit k identifikaci všech paketů přijatých ze sítě, normálně se však nepoužívá. Hashovací tabulka `ptype_base` je hashována identifikátorem protokolu a slouží k rozhodování, kterému protokolu se má předat došlý síťový paket. Síťový bottom-half ovladač porovnává typ protokolu v došlém bufferu `sk_buff` s jednou nebo více položkami `packet_type` v tabulce. Protokolu může odpoví-

dat více než jedna položka, například při odposlechu veškerého provozu na síti, a v takovém případě se buffer `sk_buff` klonuje. Pak se buffer `sk_buff` předá obslužné rutině odpovídajícího protokolu.

11

10.5.3 Odesílání IP paketů

Pakety jsou buď odesílány síťovými aplikacemi při výměně dat, nebo je generují samotné síťové protokoly při navazování spojení a podpoře navázaného spojení. Ať už paket vznikne jakkoliv, vytvoří se pro uložení jeho dat a hlaviček přidávaných protokoly různých vrstev struktura `sk_buff`.

Strukturu `sk_buff` je třeba předat síťovému zařízení, které ji odešle. Nejprve se protokol, řekněme IP, musí rozhodnout, které síťové zařízení má použít. Záleží to na optimální trase. Pokud je počítač připojen k síti jediným modemem, například protokolem PPP, je volba trasy jednoduchá. Paket se bude buď zpětným zařízením posílat na lokální počítač, nebo se bude posílat na bránu na druhém konci modemového spojení. U počítačů připojených k síti ethernet je rozhodování složitější, protože v jednom okamžiku mají přístupných mnoho počítačů.

U každého odesílaného paketu používá protokol IP směrovací tabulky, v nichž zjišťuje trasu k cílové adrese. Každá IP adresa úspěšně nalezená ve směrovací tabulce vrací strukturu `rtable`, která popisuje trasu, jež se má použít. Sem spadá jednak zdrojová IP adresa, která se má použít, adresa datové struktury `device` síťového zařízení a případně nezbytné hardwarově závislé údaje. Tyto údaje se týkají přímo síťového zařízení a obsahují například fyzickou zdrojovou a cílovou adresu a další informace specifické pro konkrétní médium. Pokud je síťovým zařízením ethernet, pak budou hardwarové údaje vypadat stejně jako na obrázku 10.1 a zdrojová a cílová adresa budou fyzické ethernetové adresy. Hardwarová hlavička se trvale uchovává u každé trasy, protože je nutné ji připojit ke každému odesílanému paketu a její opakované sestavování by zbytečně zdržovalo. Hardwarová hlavička může obsahovat fyzické adresy, které se musejí nejprve přeložit protokolem ARP. V takovém případě bude odeslání paketu pozastaveno, dokud se nepodaří potřebné adresy zjistit. Jakmile jsou adresy jednou určeny a sestaví se hardwarová hlavička, uloží se, takže další odesílané pakety už nebudou muset protokolem ARP nic zjišťovat.

12

10.5.4 Fragmentace dat

Každé síťové zařízení má dánu maximální velikost paketu a není schopno odeslat nebo přimout paket větší. S tím IP protokol počítá a je schopen fragmentovat data na menší jednotky, aby se vyhovělo maximální velikosti paketu toho zařízení, které bude paket přenášet. Hlavička IP paketu obsahuje i fragmentační pole, v němž je uložen příznak fragmentace a fragmentační offset.

13 Když je IP paket připraven k odeslání, IP nalezne síťové zařízení, přes nějž bude paket odesílat. Toto zařízení se nalezne podle směrovacích tabulek IP protokolu. Každá položka device obsahuje mimo jiné údaj o maximální přenosové jednotce (v bajtech), údaj mtu. Pokud je *mtu* zařízení menší než velikost IP paketu, jež se má odeslat, je nutné rozdělit IP paket na menší části (o velikosti *mtu*). Každý fragment je reprezentován vlastní strukturou sk_buff, v IP hlavičce je označeno, že se jedná o fragment a je uveden jeho offset v původním paketu. Poslední fragment je označen jako poslední. Pokud v průběhu fragmentace nebude IP protokol schopen alokovat další strukturu sk_buff, odeslání se nezdaří.

Příjem IP fragmentů je poněkud složitější než jejich odesílání, protože jednotlivé fragmenty mohou být obecně přijaty v libovolném pořadí a před složením je nutné přijmout všechny. 14 Vždy, když se přijme IP paket, kontroluje se, zda se nejedná o IP fragment. Když se přijme první fragment zprávy, IP vytvoří novou datovou strukturu ipq, která se připojí k seznamu ipqueue IP fragmentů, čekajících na rekombinaci. Při příjmu dalších fragmentů se vždy nalezne správná struktura ipq a přidá se k ní nová struktura ipfrag popisující nově přijatý fragment. Každá datová struktura ipq jednoznačně určuje fragmentovanou zprávu podle zdrojové a cílové IP adresy, identifikátoru protokolu a identifikátoru IP rámce. Jakmile se přijmou všechny fragmenty, zkombinují se do jediné struktury sk_buff a postoupí se ke zpracování vyšší protokolové vrstvě. Každá struktura ipq obsahuje časovač znovu spouštěný po přijetí každého nového fragmentu. Pokud tento časovač vyprší, datová struktura ipq a její struktury ipfrag budou zrušeny a předpokládá se, že se části paketu cestou ztratily. Pak záleží na nadřízené protokolové vrstvě, aby zajistila opakované zaslání zprávy.

10.6 Protokol ARP (Address Resolution Protocol)

Protokol ARP je nástroj pro překlad IP adres na fyzické hardwarové adresy, například ethernetové adresy. Protokol IP potřebuje tento překlad předtím, než může zařízení předat paket 15 (ve formátu sk_buff) k odeslání.

Protokol IP provádí různé kontroly, aby zjistil, zda příslušné zařízení potřebuje hardwarovou hlavičku, a pokud ano, zda není nutné vybudovat hlavičku znovu. Linux ukládá hardwarové 16 hlavičky ve vyrovnávací paměti, aby je nemusel neustále opakovaně vytvářet. Pokud je nutné sestavit hardwarovou hlavičku znovu, zavolá se rutina sestavení hlavičky v ovladači příslušného síťového zařízení. Všechna ethernetová zařízení používají stejnou obecnou rutinu pro sestavení hlavičky, která používá protokol ARP k překladu cílové IP adresy na fyzickou adresu.

Protokol ARP je sám o sobě velmi jednoduchý a skládá se ze dvou typů zpráv: ARP žádosti a ARP odpovědi. ARP žádost obsahuje IP adresu kterou je třeba přeložit, a odpověď obsahuje její překlad, hardwarovou adresu. ARP žádost se hromadně zašle na všechny počítače při-

pojené k síti, takže například na ethernetové síti zachytí ARP žádost všechny počítače na stejném segmentu. Počítač s požadovanou IP adresou reaguje na žádost a pošle odpověď, v níž uvádí svou hardwarovou adresu.

Protokolová vrstva ARP je v Linuxu postavena na tabulce datových struktur arp_table, které každá popisují překlad jedné IP adresy na fyzickou adresu. Tyto položky se vytvářejí, když je potřeba přeložit IP adresu, a odstraňují se po určité době. Každá datová struktura arp_table má následující položky:

poslední použití	Čas, kdy byla tato položka naposledy použita.
poslední aktualizace	Čas, kdy byla položka naposledy aktualizována.
příznaky	Popisují stav položky, například zda je úplná a podobně.
IP adresa	IP adresa, kterou tato položka popisuje.
hardwarová adresa	Přeložená hardwarová adresa.
hardwarová hlavička	Ukazatel na uloženou hardwarovou hlavičku.
časovač	Položka timer_lost používaná k rušení ARP žádostí, na něž nepřišla odpověď.
opakování	Počet pokusů, kolikrát se vznášela žádost.
fronta sk_buff	Seznam struktur sk_buff, které čekají na překlad této IP adresy.

ARP tabulka se skládá z tabulky ukazatelů (vektor arp_tables) na řetězce položek arp_table. Položky se kvůli zrychlení přístupu ukládají tak, že poslední dva bajty IP adresy generují index do tabulky a pak se prohlíží řetězec položek, až se najde ta správná. Linux ukládá připravené hardwarové hlavičky rovněž ve struktuře hh_cache, do níž struktura arp_table ukazuje.

Když je požadován překlad IP adresy a v ARP tabulce není požadovaná položka nalezena, musí ARP poslat ARP žádost. Vytvoří se nová položka arp_table a do její fronty se zařadí sk_buff obsahující paket, který překlad potřebuje. Odešle se ARP žádost a spustí se časovač. Pokud se do zadaného intervalu neobjeví odpověď, ARP několikrát žádost zopakuje a pokud odpověď stále nepřichází, odstraní se položka arp_table. Na tuto skutečnost budou upozorněny všechny struktury sk_buff, které čekaly ve frontě zrušené položky, a záleží na protokolové vrstvě, která je chtěla odeslat, jak se s tím vyrovná. UDP se o ztracené pakety nestará, TCP se pokusí o opakované odeslání. Pokud vlastník požadované IP adresy odpoví svou hardwarovou adresou, položka arp_table tabulky se označí jako úplná a všechny struktury sk_buff v její frontě budou předány k odeslání. Hardwarová adresa se zapíše do hardwarové hlavičky každého bufferu sk_buff.

Protokolová vrstva ARP musí také odpovídat na ARP žádosti na IP adresu lokálního počítače. Registruje si svůj protokol (ETH_P_ARP), čímž se generuje datová struktura packet_type. Znamená to, že se jí budou předávat všechny přijaté ARP pakety. Kromě ARP odpovědí tedy vrstva obdrží i všechny ARP žádosti. Sama generuje ARP odpovědi podle hardwarové adresy uložené ve struktuře device toho zařízení, které žádost přijalo.

S postupem času se topologie sítě může měnit a IP adresy mohou být přiřazovány jiným hardwarovým adresám. Například některé vytáčené služby přiřazují IP adresy vždy při navázání spojení. Aby ARP tabulka obsahovala aktuální informace, spouští ARP periodický časovač, který prohlédne všechny struktury arp_table a hledá a odstraňuje prošlé. Zároveň ovšem hlídá, aby neodstranil položky, kterým je přiřazena hardwarová hlavička. Odstranění takových položek by mohlo být nebezpečné, protože na nich závisí další datové struktury. Některé položky ARP tabulky jsou trvalé a jsou označeny tak, že nebudou nikdy zrušeny. Vždy, když se alokuje nová položka a ARP tabulka by dosáhla své maximální velikosti, tabulka se redukuje vyhledáním a zrušením nejstarších položek.

10.7 Směrování

Směrování rozhoduje o tom, kam poslat IP pakety určené konkrétní IP adrese. Při odesílání IP paketu je mnoho rozhodování. Dá se cíle vůbec dosáhnout? Pokud ano, kterým síťovým zařízením paket odeslat? Pokud je možno použít více síťových zařízení, které je nejlepší? Odpovědi na tyto otázky poskytuje směrovací tabulka. Jedná se o dvě databáze, důležitější je *databáze směrovacích informací*. Jedná se o úplný seznam známých IP cílů a nejlepších tras k nim. Druhá, menší a rychlejší databáze, *směrovací cache*, slouží k rychlému nalezení tras na různé cíle. Stejně jako všechny vyrovnávací paměti, i ona obsahuje pouze nejčastěji používané trasy, její obsah se odvozuje z databáze směrovacích informací.

Trasy se přidávají a ruší na základě IOCTL žádostí soketového rozhraní BSD. Tyto žádosti se předávají ke zpracování příslušnému protokolu. Protokolová vrstva INET povoluje rušit a přidávat IP trasy pouze procesům s oprávněními superuživatele. Trasy mohou být pevné nebo se mohou v čase dynamicky měnit. Většina systémů používá pevné trasy, dynamické trasy obvykle používají pouze routery. Router používá směrovací protokoly, jimiž trvale ověřuje dostupnost tras ke všem známým IP cílům. Systémy, které nepracují jako routery, se označují jako koncové systémy. Směrovací protokoly jsou implementovány jako démony, například *gated*, a trasy přidávají a ruší rovněž přes funkci IOCTL soketového rozhraní BSD.

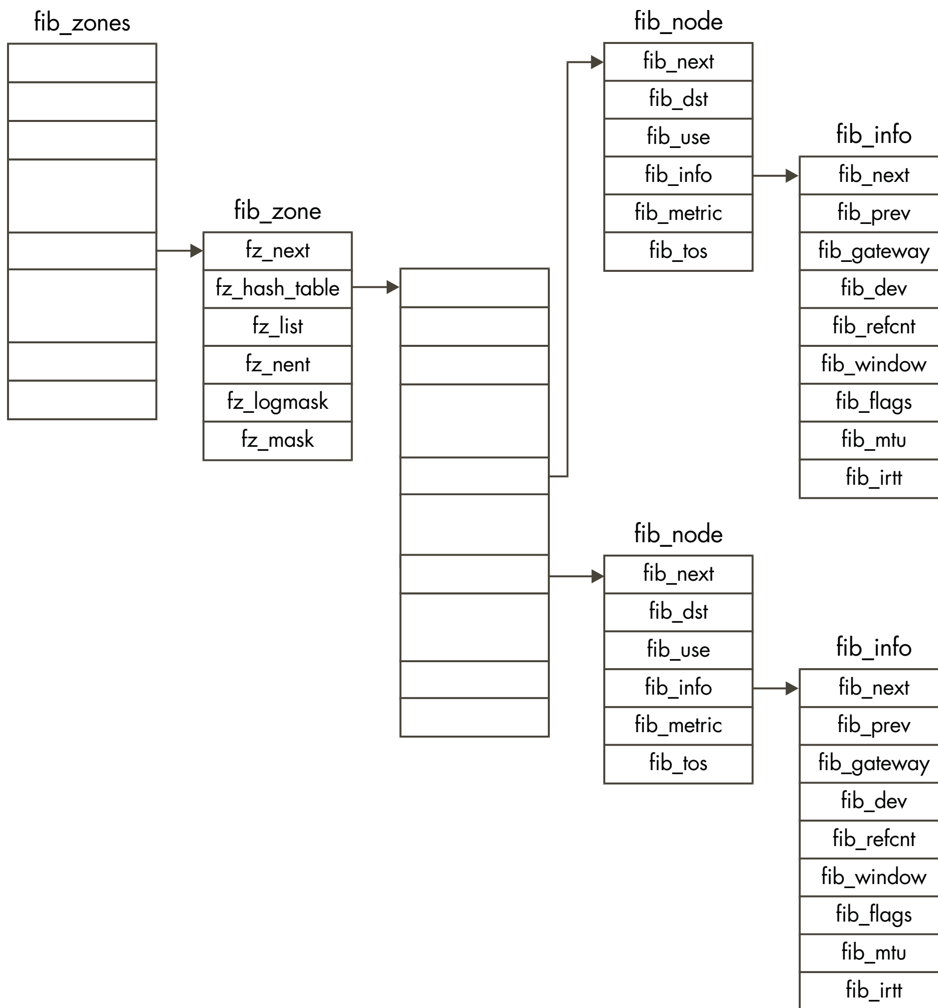
10.7.1 Směrovací cache

Vždy, když se hledá trasa, prohlíží se nejprve směrovací cache. Pokud v ní není požadovaná trasa nalezena, hledá se trasa v databázi směrovacích informací. Pokud trasa není ani tam, není možno paket odeslat a aplikace na to bude upozorněna. Pokud je trasa v databázi směrovacích informací a není ve směrovací cache, vytvoří se pro ni nová položka a přidá se do cache. Směrovací cache je tabulka (`ip_rt_hash_table`), která obsahuje ukazatele na řetězce struktur `rtable`. Index do směrovací tabulky je hashovací funkce založená na dvou nejméně významných bajtech IP adresy. U těchto dvou bajtů je nejpravděpodobnější, že se budou pro různé cíle lišit, a tak se zajišťuje nejlepší rozptyl hodnot v hashovací tabulce. Každá struktura `rtable` obsahuje informace o trase: cílovou IP adresu, zařízení používané pro směrování k danému cíli, maximální povolenou velikost paketu tohoto zařízení a další. Obsahuje také referenční počitadlo, počitadlo použití a časovou značku doby posledního použití. Referenční počitadlo se inkrementuje vždy, když je trasa používána, aby se zaznamenával počet síťových spojení, které danou trasu používají. Hodnota se dekrementuje, když zařízení přestane trasu používat. Počitadlo použití se inkrementuje při každém hledání trasy a používá se k seřazování položek `rtable` v jednotlivých řetězcích hashovací tabulky. Periodicky se kontroluje časová značka jednotlivých položek a zjišťuje se, zda položka není příliš stará. Pokud trasa nebyla delší dobu použita, odstraní se z vyrovnávací paměti. Trasy jsou ve vyrovnávací paměti uloženy tak, že nejčastěji používané trasy stojí na počátku hashovacích řetězců. Díky tomu je jejich nalezení nejrychlejší.

10.7.2 Databáze směrovacích informací

Databáze směrovacích informací (viz obr. 10.5) obsahuje přehled všech tras, které systém v daném okamžiku zná. Jedná se o poměrně komplikovanou datovou strukturu a i když je uspořádána poměrně efektivně, rozhodně se v ní nehledá příliš rychle. Bylo by každopádně velmi pomalé hledat v této databázi trasu pro každý odesílaný IP paket. Z toho důvodu se používá směrovací cache, která zrychluje odesílání IP paketu na často používané adresy. Směrovací cache se odvozuje z databáze směrovacích informací a obsahuje její nejčastěji používané položky.

Každá IP podsíť je reprezentována datovou strukturou `fib_zone`. Na tyto struktury se ukazuje z hashovací tabulky `fib_zones`. Hashovací index se odvozuje z masky IP podsítě. Všechny trasy na stejné podsítě jsou popsány párem struktur `fib_node` a `fib_info` uložených ve frontě `fz_list` každé datové struktury `fib_zone`. Pokud v jedné podsíti bude velké množství tras, vygeneruje se hashovací tabulka, která usnadní hledání struktur `fib_node`.



Obrázek 10.5
Databáze směrovacích informací

Na stejnou podsíť může existovat několik tras, které vedou přes různé brány. Směrovací vrstva IP protokolu nepovoluje více tras na jednu podsíť přes stejnou bránu. Jinak řečeno, pokud na jednu podsíť vede více tras, každá vede přes jinou bránu. Každá trasa má přiřazenu svou *metriku*. Metrika udává míru výhodnosti trasy. V zásadě představuje metrika trasy počet podsítí, přes které se musí projít, než se dojde k požadovanému cíli. Čím vyšší metrika, tím horší trasa.

Odkazy na zdrojové texty jádra

1 – Viz include/-linux/net.h
2 – Viz include/-net/sock.h
3 – Viz sys_socket() in net/socket.c
4 – Viz include/-linux skbuff.h
5 – Viz skb_push() in include/-linux/skbuff.h
6 – Viz skb_pull() in include/-linux skbuff.h
7 – Viz skb_put() in include/linux/-skbuff.h
8 – Viz skb_trim() in include/-linux/skbuff.h
9 – Viz netif_rx() in net/core/dev.c
10 – Viz net_bh() in net/core/dev.c
11 – Viz ip_recv() in net/ipv4/-ip_input.c
12 – Viz include/-net/route.h
13 – Viz ip_build_xmit() in net/ ipv4/ip_output.c
14 – Viz ip_rcv() in net/ipv4/-ip_input.c
15 – Viz ip_build_xmit() in net/ipv4/-ip_output.c
16 – Viz rebuild_header()in net/-ethernet/eth.c
17 – Viz ip_rt_check_expire() in net/ipv4/-route.c

Kapitola

11

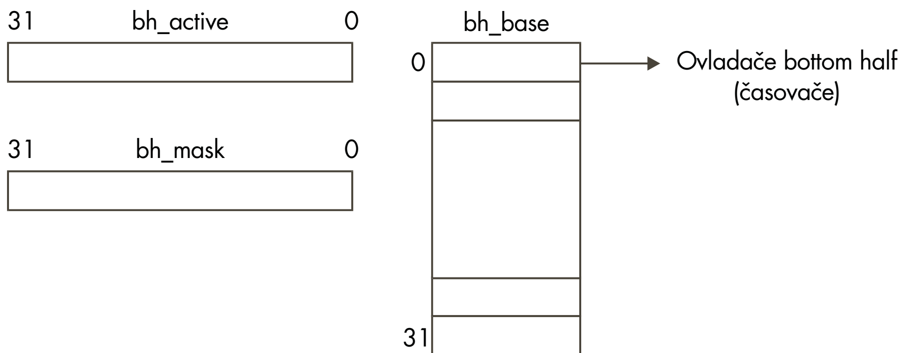
Mechanismy jádra

V této kapitole jsou popsány některé obecné činnosti a mechanismy, které musí jádro Linuxu zajišťovat, aby mohly jednotlivé jeho části efektivně spolupracovat.

11.1 Obslužný mechanismus „bottom-half“

Velmi často se v jádře stává, že nějakou činnost nechcete vykonat okamžitě. Typickým příkladem je zpracování přerušení. Když dojde k přerušení, procesor přeruší právě prováděnou činnost a operační systém doručí obsluhu přerušení příslušnému ovladači zařízení. Ovladače ale nemohou strávit obsluhou příliš mnoho času, protože v době obsluhy přerušení nemůže v systému běžet nic jiného. Často by se při obsluze prováděla činnost, kterou je možno stejně dobře vykonat až někdy později. Obslužný mechanismus bottom-half byl vyvinut právě k tomu, aby ovladače zařízení a další části jádra mohly naplánovat pozdější provedení nějaké činnosti. Na obrázku 11.1 jsou znázorněny datové struktury, které s tímto typem obsluhy souvisejí.

Může existovat až 32 bottom-half handlerů, `bh_base` je vektor ukazatelů na obslužné rutiny jednotlivých handlerů. Struktury `bh_active` a `bh_mask` mají příslušné bity nastaveny podle toho, které handlery byly instalovány a jsou aktivní. Pokud je nastaven N-tý bit struktury `bh_mask`, znamená to, že N-tý prvek pole `bh_base` obsahuje platnou adresu obslužné rutiny. Pokud je nastaven N-tý bit masky `bh_active`, znamená to, že obslužná rutina by měla být volána jakmile to plánovač uzná za vhodné. Tyto indexy jsou definovány staticky. Bottom-half handler časovače má nejvyšší prioritu (index 0), následuje bottom-half handler konzoly (s indexem 1) a tak dále.Typicky mají tyto obslužné rutiny k sobě přiřazen seznam úloh, které je zapotřebí provést. Například bottom-half handler *immediate* zpracovává úkoly z fronty akutních úloh (`tq_immediate`), která obsahuje úkoly, jež je nutno provést co nejdříve.



Obrázek 11.1
Datové struktury bottom-half obsluhy

Některé bottom-half handlery v jádře jsou specifické podle zařízení, další jsou ale obecnější:

TIMER Tento handler se aktivuje vždy, když se objeví přerušení od systémového časovače, a slouží k obsluze front časovačů v jádře.

CONSOLE Tento handler zpracovává zprávy konzoly.

TQUEUE Tento handler zpracovává zprávy od zařízení `tty`.

NET Tento handler provádí obecnou obsluhu sítí.

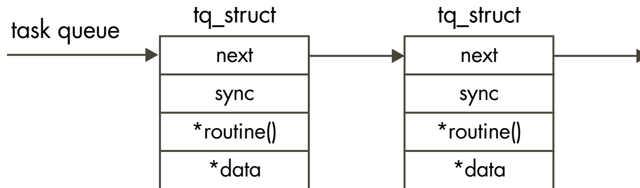
IMMEDIATE Obecný handler používaný různými ovladači zařízení k provádění činností, odložených na později.

Kdykoliv ovladač zařízení nebo nějaká jiná část jádra potřebuje naplánovat nějakou činnost na později, přidá úkol do vhodné systémové fronty, například do fronty časovače, a pak signalizuje jádru, že je třeba provést nějakou bottom-half obsluhu. Dosáhne se toho nastavením příslušného bitu v `bh_active`. Bit 8 se nastavuje v případě, že ovladač zařadil nějaký úkol do fronty handleru *immediate* a přeje si, aby byl handler *immediate* spuštěn a provedl úkol. Bitová mapa `bh_active` se kontroluje na konci každého systémového volání, těsně před předáním řízení zpět volajícímu procesu. Pokud má nějaký bit nastaven, volají se obslužné rutiny aktivovaných bottom-half handlerů. Nejprve se kontroluje bit 0, poté bit 1 a tak dále až k bitu 31.

Bit ve struktuře `bh_active` se nuluje po volání příslušného bottom-half handleru. Struktura `bh_active` je transientní, má význam pouze mezi voláními plánovače a představuje metodu, jak nevolat bottom-half handlery v případě, kdy pro ně není žádná práce.

11.2 Fronty úloh

Fronty úloh představují způsob, kterým jádro odkládá různé činnosti na pozdější dobu. Linux má obecné mechanismy pro řazení úloh do front a jejich pozdější provedení.



Obrázek 11.2
Fronta úloh

Fronty úloh se často používají společně s bottom-half obsluhou, například při volání bottom-half obsluhy časovače se zpracovává fronta úloh časovače. Fronta úloh je jednoduchá datová struktura, jak vidíme na obrázku 11.2. Skládá se z lineárního seznamu struktur `tq_struct`, z nichž každá obsahuje adresu rutiny a ukazatel na nějaká data.

Rutina se bude volat při zpracovávání daného prvku fronty a předá se jí ukazatel na data.

Jakákoliv část jádra, například ovladač zařízení, může vytvářet a používat fronty úloh, kromě toho existují tři fronty úloh vytvořené a udržované jádrem:

timer Fronta časovače slouží k řazení úloh, které se provedou s příštím tikem systémových hodin. Při každém tiku se tato fronta kontroluje, zda obsahuje nějaké položky, a pokud ano, aktivuje se bottom-half handler časovače. Bottom-half handler časovače se volá, stejně jako ostatní bottom-half handlery, při příštím běhu plánovače. Nezaměňujte tuto frontu se systémovými časovači, které představují podstatně inteligentnější mechanismus.

immediate Fronta akutních úloh se rovněž zpracovává když plánovač aktivuje bottom-half handlery. Handler obsluhy akutní fronty má nižší prioritu než obsluha časovače, takže úlohy v této frontě budou provedeny později.

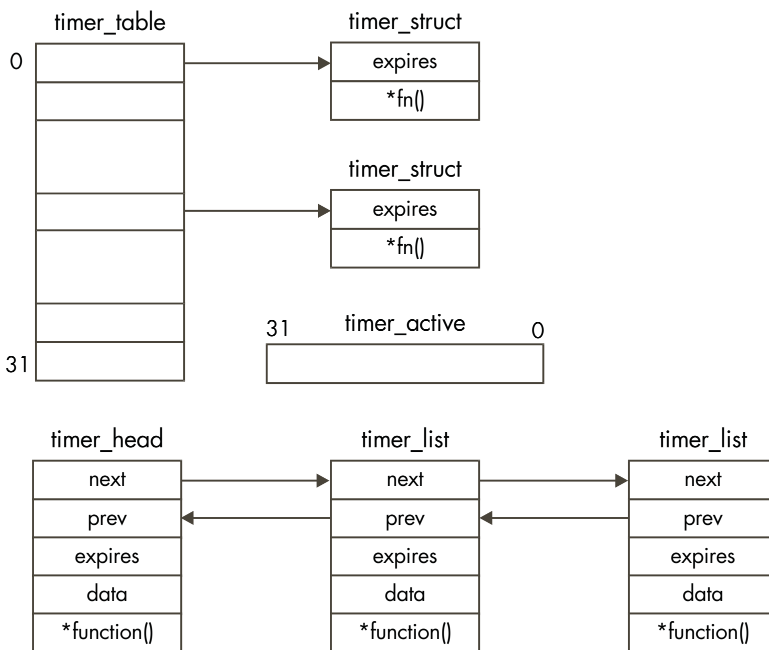
scheduler Frontu plánovače zpracovává přímo plánovač. Slouží k podpoře dalších front úloh v systému a řadí se do ní rutiny, které zpracovávají ostatní fronty úloh, například fronty úloh vytvářené ovladači zařízení.

Při zpracovávání fronty úloh se nejprve zruší ukazatel na první prvek fronty a přiřadí se mu hodnota NULL. Toto odstranění je atomická operace, tedy operace, kterou není možno přerušit. Následně se volá obslužná rutina každého prvku fronty. Prvky fronty často bývají static-

ky alokovaná data. Neexistuje však žádný obecný mechanismus pro uvolnění alokované paměti. Obsluha fronty se pouze přesouvá z jednoho prvku na druhý. Korektní uvolnění alokované paměti jádra musí zajistit přímo volaná rutina každé položky.

11.3 Časovače

Operační systém musí být schopen naplánovat provedení nějaké úlohy někdy v budoucnu. Je nutný mechanismus, který zajistí, aby se daná úloha provedla někdy v budoucnu v relativně přesně určeném okamžiku. Každý mikroprocesor musí být vybaven programovatelným intervalovým časovačem, který pravidelně přerušuje procesor. Toto pravidelné přerušení se označuje jako systémové hodiny a funguje trochu jako metronom, kdy udává rytmus systémových aktivit.



Obrázek 11.3
Systémové časovače

Linux má velmi jednoduchou představu o čase, měří čas v počtu systémových tiků, k nimž došlo od zavedení systému. Všechny časy v systému se měří v těchto jednotkách, které se označují jako `jiffies` podle globálně přístupné proměnné stejného jména.

Linux používá dva typy systémových časovačů, oba řadí do front úkoly, které se mají provést v nějakém čase, jejich implementace se však mírně liší. Na obrázku 11.3 jsou znázorněny oba tyto mechanismy.

První, starší mechanismus obsahuje statické pole 32 ukazatelů na datové struktury `timer_struct` a masku aktivních časovačů, `timer_active`. Řazení časovačů v tabulce časovačů je staticky definováno (podobně jako v tabulce `bh_base` bottom-half obsluhy). Většina položek se do této tabulky přidává v době inicializace systému. Druhý, novější mechanismus používá seznam datových struktur `timer_list`, které jsou řazeny vzestupně podle doby, kdy příslušné časovače vyprší.

4

Obě metody měří čas k vypršení časovače v `jiffies`, takže časovač, který chce být aktivován za pět sekund, musí nejprve převést pět sekund na `jiffies` a přičíst získanou hodnotu k aktuálnímu systémovému času, čímž získá čas, v němž má vypršet. S každým tikem systémových hodin se aktivuje bottom-half handler časovače, takže při příštím běhu plánovače budou zpracovány položky ve frontě úloh časovače. Bottom-half handler časovače zpracovává časovače obou typů.

U starších časovačů se kontroluje bitová maska `timer_active` aby se zjistilo, zda je příslušný časovač aktivní. Pokud daný časovač vypršel (jeho čas vypršení je menší než aktuální hodnota `jiffies`), volá se jeho obslužná rutina a vynuluje se příznak jeho aktivity.

5

6

Při obsluze novějších typů časovačů se kontrolují položky v seznamu struktur `timer_list`. Každý vypršený časovač se odstraní ze seznamu a zavolá se jeho obslužná rutina. Novější mechanismus má tu výhodu, že obslužné rutině časovače je možno předávat parametry.

7

11.4 Čekací fronty

Často se stává, že proces musí čekat na nějaký systémový prostředek. Proces například potřebuje ke své práci VFS inode popisující nějaký adresář souborového systému, daný inode však není přítomen ve vyrovnávacích pamětech. V takovém případě musí proces počkat, dokud se inode nenačte z fyzického média obsahujícího daný souborový systém.

wait_queue
*task
*next

Obrázek 11.4
Čekací fronta

Linux používá jednoduchou datovou strukturu, čekací frontu (viz obrázek 11.4), která se skládá z ukazatele na strukturu `task_struct` procesu a z ukazatele na další prvek v čekací frontě.

Proces přidaný na konec čekací fronty může být buď přerušitelný, nebo nepřerušitelný. Přerušitelné procesy mohou být v čekání přerušeny událostmi jako vypršení časovače nebo doručení signálu. Tato vlastnost se odráží ve stavu čekajícího procesu, který může být buď `INTERRUPTIBLE`, nebo `UNINTERRUPTIBLE`. Jakmile je proces přidán do čekací fronty, nemůže dále pokračovat v práci a spouští se plánovač, který naplánuje běh jiného procesu.

Při zpracovávání čekací fronty se stav každého procesu mění na `RUNNING`. Pokud byl proces odstraněn z fronty běžících procesů, opět se do ní přidá. Při příštím běhu plánovače se kandidáty na spuštění stávají i procesy v čekací frontě, protože nyní už nečekají. Když se naplánuje spuštění procesu v čekací frontě, proces jako první odstraní sám sebe z čekací fronty. Čekací fronty se používají například k synchronizaci přístupu k systémovým prostředkům a používají se rovněž při implementaci semaforů (viz dále).

11.5 Zámky

Zámky (*buzz locks* nebo *spin locks*) představují jednoduchou metodu ochrany datových struktur nebo částí kódu. Umožňují, aby v daném okamžiku vykonával kritickou sekci kódu pouze jediný proces. Linux je používá při omezení přístupu k položkám svých datových struktur, jako zámek slouží jedna položka struktury. Vždy, když chce nějaký proces změnit nějakou část datové struktury, změní hodnotu zámku z 0 na 1. Pokud je hodnota rovna 1, proces ji stále testuje a běhá v krátké testovací smyčce. Přístup k paměťové oblasti, v níž je zámek uložen, musí být atomickým, operace čtení hodnoty zámku, testování, zda je nulová, a nastavení na jedničku nesmí být přerušitelná žádným jiným procesem. Většina procesorových architektur nabízí speciální instrukce pro podporu takovýchto operací, zámky je však možno implementovat i s využitím necachované hlavní paměti.

Když proces opouští kritickou sekci kódu, dekrementuje zámek a vrací tak jeho hodnotu zpět na nulu. Pokud nějaký proces právě běhá v čekací smyčce na kritickou sekci, zjistí, že hodnota je nyní nulová, inkrementuje ji na jedničku a vstoupí do kritické sekce sám.

11.6 Semafory

Semafory slouží k ochraně kritických sekcí kódu nebo datových struktur. Připomeňme si, že všechny přístupy ke kritickým částem dat, například k inodům VFS, provádí jádro jménem nějakého procesu. Bylo by velmi nebezpečné, kdyby nějaký proces mohl měnit kritická data, používaná právě jiným procesem. Jedna metoda ochrany by mohla spočívat v použití zámků

kolem kritických dat, jedná se však o velmi prostou metodu, která by nevedla k optimálnímu výkonu systému. Namísto toho používá Linux semafory, které umožňují přístup ke kritickým částem kódu a dat vždy jen jednomu procesu, ostatní procesy, které požadují přístup ke stejnému prostředku, budou čekat až do doby, než se prostředek uvolní. Čekající procesy jsou pozastaveny, takže v běhu mohou pokračovat jiné procesy.

Datová struktura `semaphore` v Linuxu obsahuje následující informace:

count Tato položka udává počet procesů, které mohou daný prostředek použít. Kladná hodnota znamená, že prostředek je možno použít. Záporná nebo nulová hodnota znamená, že nějaký proces na prostředek čeká. Počáteční hodnota 1 znamená, že daný prostředek může použít právě jeden proces. Když proces požaduje prostředek, dekrementuje tuto hodnotu, a když skončí s jeho použitím, inkrementuje ji.

waking Počet procesů čekajících na daný prostředek, tedy počet procesů, které je třeba probudit, jakmile se prostředek uvolní.

wait queue Když proces čeká na prostředek, je přesunut do čekací fronty.

lock Zámek používaný při přístupu k položce `waking`.

Předpokládejme, že počáteční hodnota semaforu je 1. První proces, který k němu dorazí, zjistí, že počitadlo je kladné a dekrementuje je o jedničku, nová hodnota tedy bude 0. Proces nyní „vlastní" kritickou část kódu nebo prostředek, chráněný semaforem. Když proces kritickou sekci opouští, inkrementuje počitadlo semaforu. Nejoptimálnější případ nastává tehdy, pokud v dané chvíli žádný jiný proces nečeká na vlastnictví chráněné sekce. Semafory jsou v Linuxu optimalizovány tak, aby pracovaly nejefektivněji právě v tomto, nejčastějším případě.

Pokud chce vlastnictví kritické sekce, právě vlastněné nějakým procesem, získat další proces, rovněž dekrementuje počitadlo. Hodnota počitadla bude nyní záporná (-1) a proces nemůže vstoupit do kritické sekce. Namísto toho musí počkat, dokud ji vlastník neopustí. Čekající proces se přidá do čekací fronty semaforu a cyklicky testuje hodnotu `waking` s tím, že dokud tato hodnota nebude nenulová, předává vždy řízení plánovači.

Vlastník kritické sekce inkrementuje počitadlo semaforu a pokud je nová hodnota menší nebo rovna nule, pak existuje nějaký spící proces, čekající na prostředek. V nejlepším případě bude nová hodnota semaforu rovna jedné a žádná další činnost není zapotřebí. V opačném případě vlastník inkrementuje počitadlo `waking` a probudí proces čekající ve frontě semaforu. Když se proces probudí, zjistí, že hodnota počitadla `waking` je jedna a ví, že může vstoupit do kritické sekce. Dekrementuje hodnotu `waking`, vrátí ji tedy zpět na nulu, a pokračuje. Veškerý přístup k položce `waking` semaforu je chráněn zámkem v položce `lock` semaforu.

Odkazy na zdrojové texty jádra

1 – Viz include/linux/-interrupt.h
2 – Viz do_bottom_half() in kernel/-softirq.c
3 – Viz include/-linux/tqueue.h
4 – Viz include/-linux/timer.h
5 – Viz timer_bh() in kernel /sched.c
6 – Viz run_old_timers)) in kernel/sched.c
7 – Viz run_timer_list() in kernel/sched.c
8 – Viz include/-linux/wait.h
9 – Viz include/-asm/semaphore.h

Kapitola

12

Moduly

V této kapitole je popsáno, jak může jádro Linuxu dynamicky nahrávat různé funkce, například souborové systémy, pouze v případě, že je potřebuje.

Jádro Linuxu je monolitické, jedná se tedy o jediný velký program, v němž mají všechny komponenty přístup ke všem interním datovým strukturám a rutinám. Alternativním řešením by bylo použití mikrojádra, kdy by byly jednotlivé funkční celky jádra rozděleny do samostatných jednotek, které by spolu mohly komunikovat pouze přísně omezenými mechanismy. Takto by ale bylo přidávání nových komponent do jádra v době jeho konfigurace časově velmi náročné. Řekněme, že chcete použít ovladač SCSI pro řadič SCSI NCR 810 a nemáte jej vestavěn přímo v jádře. Museli byste nakonfigurovat a sestavit nové jádro s tímto ovladačem. Linux ale umožňuje dynamicky nahrávat a rušit komponenty operačního systému podle potřeby. Moduly Linuxu představují části kódu, které je možno dynamicky přilinkovat do jádra v kterémkoliv okamžiku po zavedení systému. Je možno je z jádra zrušit a odstranit ve chvíli, kdy je není déle potřeba. Většinou se jako moduly vytvářejí ovladače zařízení, ovladače pseudozařízení (například ovladače sítí) a souborové systémy.

Moduly jádra můžete nahrávat a rušit buď explicitně pomocí příkazů `insmod` a `rmmod`, nebo si jejich nahrání či odstranění může vynutit jádro prostřednictvím démona jádra (`kerneld`).*

Dynamické nahrávání kódu podle potřeby je velmi lákavé, protože umožňuje udržovat minimální velikost jádra se zachováním vysoké flexibility. V jádře mého Linuxu používám moduly poměrně často, a tak je veliké pouze 406 KB. Občas používám souborový systém `VFAT`, takže mám jádro nakonfigurováno tak, aby automaticky nahrálo modul souborového systému `VFAT` v okamžiku, kdy tento systém připojím. Když oblast `VFS` odpojím, systém

* Poznámka korektora: V verzích jádra 2.1.x je démon `kerneld` nahrazen vláknem `kmod`.

zjistí, že modul souborového systém VFAT není déle potřebný a odstraní jej. Moduly mohou být velmi užitečné také při testování nových částí kódu jádra, aniž by bylo nutné jádro znovu sestavit a znovu zavést po každé změně. Nic ale není zadarmo, takže moduly jádra s sebou přinášejí lehké snížení výkonu a zvýšení paměťové náročnosti. Nahratelný modul musí obsahovat určitý speciální kód a tento kód spolu se speciálními datovými strukturami jádra má za následek zvýšení paměťové náročnosti. Kromě toho používají moduly nepřímé přístupy, takže použití prostředků jádra z modulů je méně efektivní.

Jakmile dojde k nahrání modulu, stává se částí jádra stejně, jako veškerý ostatní kód jádra. Má stejná práva a zodpovědnost jako ostatní části jádra. Jinak řečeno, modul jádra může zhroutit celé jádro stejně, jako kterákoliv jiná část kódu jádra nebo ovladače zařízení.

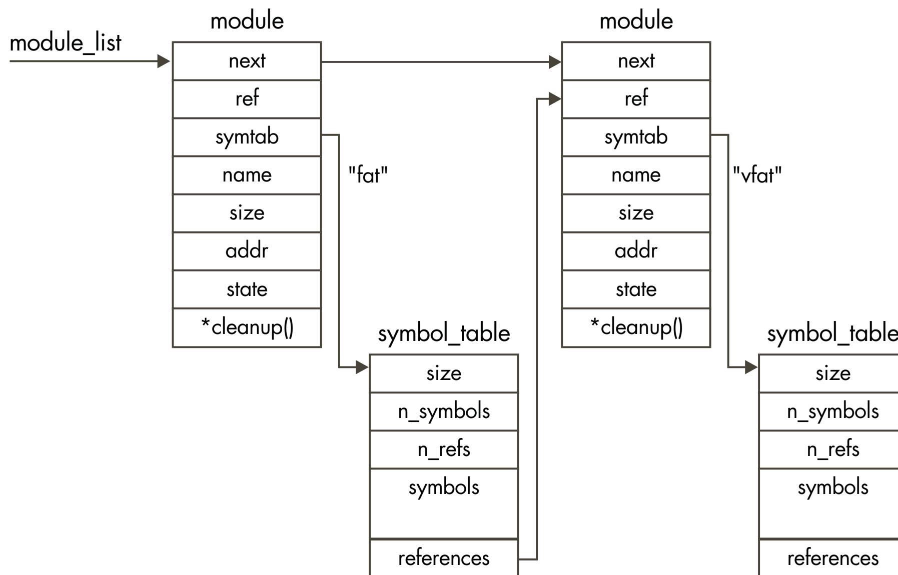
Aby mohl modul používat potřebné prostředky jádra, musí je být schopen najít. Řekněme, že modul potřebuje volat rutinu `kmalloc()`, alokační rutinu paměti jádra. V době svého sestavení modul neví, kde v paměti je rutina `kmalloc()` umístěna, takže po nahrání modulu musí jádro upravit všechny odkazy na rutinu `kmalloc()` v modulu, tak aby modul mohl pracovat. Jádro si udržuje seznam všech prostředků jádra v tabulce symbolů jádra, takže je schopno řešit odkazy na tyto prostředky ze všech nahraných modulů. Linux povoluje vrstvení modulů, tedy situaci, kdy jeden modul používá prostředky druhého. Například souborový systém VFAT potřebuje služby modulu souborového systému FAT, protože systém VFAT je víceméně pouze rozšířením služeb systému FAT. Když modul požaduje služby nebo prostředky jiného modulu, je to podobné situaci, kdy modul požaduje prostředky a služby samotného jádra. Rozdíl je pouze v tom, že v této situaci je požadovaná služba přítomna v jiném, dříve nahraném modulu. Vždy při nahrání modulu modifikuje jádro tabulku symbolů jádra a přidává do ní všechny prostředky nebo symboly exportované nově nahraným modulem. Znamená to tedy, že jakmile se nahraje další modul, má přístup ke službám všech dříve nahraných modulů.

Když je vznesen požadavek na odstranění modulu, musí jádro vědět, že modul je nepoužívaný a musí být schopno modulu sdělit, že bude odstraněn. Díky tomu bude modul před svým zrušením schopen uvolnit všechny jím alokované systémové prostředky, například paměť jádra nebo přerušení. Po zrušení modulu odstraní jádro z tabulky symbolů jádra všechny symboly exportované právě zrušeným modulem.

Kromě toho, že jakýkoliv špatně napsaný modul může zhroutit operační systém, hrozí zde i další nebezpečí. Co se stane, pokud nahrajete modul určený pro starší nebo novější verzi jádra, než kterou právě používáte? Může to způsobit problémy například pokud modul bude volat nějakou rutinu jádra a předá jí špatné parametry. Jádro se proti tomu dokáže bránit pomocí kontroly verze při nahrávání modulu.

12.1 Nahrání modulu

Existují dvě možnosti, jak může dojít k nahrání modulu jádra. První způsob používá příkaz `insmod`, který způsobí zavedení modulu do jádra. Druhý, chytřejší způsob, spočívá v nahrávání modulů podle potřeby, této metodě se říká vynucené nahrávání.



Obrázek 12.1
Seznam modulů jádra

Když jádro zjistí, že potřebuje nahrát nějaký modul, například pokud uživatel připojí souborový systém, který v jádře není, požádá jádro démona jádra (`kerneld`) o nahrání příslušného modulu.

Démon jádra je normální uživatelský proces ovšem s privilegii superuživatele. Po svém spuštění (obvykle v době zavádění jádra) otevře kanál IPC mezi sebou a jádrem. Tento kanál jádro používá k zasílání zpráv démonu `kerneld`, jimiž požaduje provedení různých operací.

Hlavním úkolem démona `kerneld` je nahrávání a rušení modulů jádra, dokáže však i jiné věci, například v případě potřeby otevřít linku PPP přes sériový kabel a posléze ji zase zavřít. Démon `kerneld` tyto úkoly neprovádí přímo, namísto toho volá příslušné příkazy, například `insmod`. Démon `kerneld` je čistě zástupce jádra, který jeho jménem provádí různé úkony.

Utilita `insmod` musí nejprve nalézt modul, který se má nahrát. Vynuceně nahrávané moduly jsou obvykle uloženy v adresáři `/lib/modules/verze jádra`. Moduly jádra jsou linkované objektové soubory stejně jako ostatní programy v systému s tou výjimkou, že jsou

1

2

relokovatelné. Znamená to, že obraz není slinkován tak, aby musel být nahrán na určitou adresu. Mohou být uloženy v objektovém formátu `a.out` nebo `elf`. Pomocí privilegovaných systémových volání zjišťuje utilita `insmod` jádrem exportované symboly.

Tyto symboly jsou organizovány v párech jména symbolu a jeho hodnoty, například adresa. Tabulka symbolů exportovaných jádrem je udržována v první datové struktuře `module` seznamu modulů, kterou udržuje jádro a ukazuje na ni ukazatel `module_list`.

Do této tabulky, která se vytváří při překladu a linkování jádra, jsou zařazovány pouze vybrané symboly jádra, jádro neexportuje modulům všechny své symboly. Příkladem může být symbol „`request_irq`“, což je rutina jádra, kterou modul musí volat, pokud chce převzít obsluhu nějakého přerušení. V mém konkrétním jádře má tento symbol hodnotu *0x0010cd30*. Jádrem exportované symboly a jejich moduly můžete zjistit výpisem souboru `/proc/ksyms` nebo příkazem `ksyms`. Utilita `ksyms` může zobrazit buď všechny jádrem exportované symboly, nebo pouze symboly exportované nahranými moduly.

Utilita `insmod` nahraje modul do své virtuální paměti a upraví všechny odkazy na rutiny a prostředky jádra pomocí tabulky symbolů, exportovaných jádrem. Tato úprava se provádí přímo přepisem obrazu modulu v paměti, utilita `insmod` fyzicky zapisuje adresy jednotlivých symbolů na příslušná místa v kódu modulu.

Když `insmod` upraví odkazy v modulu, požádá jádro o přidělení dostatečného místa pro uložení nového modulu, opět pomocí privilegovaného systémového volání. Jádro provede alokaci nové datové struktury `module` a dostatečné paměti pro uložení modulu. Strukturu `module` umístí na konec seznamu nahraných modulů jádra. Nový modul je označen jako `UNINITIALIZED`.

Na obrázku 12.1 vidíme seznam modulů jádra poté, co byly do jádra nahrány dva moduly, `FAT` a `VFAT`. Na obrázku není znázorněn první modul v seznamu, což je pseudomodul používaný pouze k uložení tabulky symbolů exportovaných jádrem. Pomocí příkazu `lsmod` můžete vypsat seznam všech modulů nahraných v jádře a jejich vzájemných závislostí. Příkaz `lsmod` pouze přeformátuje výpis souboru `/proc/modules`, který se sestavuje ze seznamu struktur `module` v jádře. Paměť alokovaná jádrem se namapuje do adresového prostoru procesu `insmod`, takže do ní proces může přistupovat. Utilita `insmod` zkopíruje modul do alokovaného prostoru a provede jeho relokaci tak, aby běžel od té adresy, kterou mu jádro přidělilo. Tento postup je nutný, protože modul nemůže počítat s tím, že se bude dvakrát nahrávat na stejnou adresu, nebo že jej nahrají na stejnou adresu dva různé systémy. Relokace opět spočívá ve fyzické úpravě adres a odkazů v modulu.

Nový modul také exportuje své symboly, takže `insmod` musí sestavit tabulku těchto symbolů. Každý modul jádra musí obsahovat svůj inicializační a ukončovací kód, jejichž adresy se neexportují, `insmod` je ale musí znát a sdělit jádru. Pokud šlo všechno dobře, může nyní `insmod` provést inicializaci modulu, která se provádí privilegovaným systémovým voláním, jímž se jádru předávají adresy inicializační a ukončovací rutiny modulu.

Po přidání nového modulu do jádra je nutné aktualizovat tabulku symbolů jádra a upravit moduly, které nový modul používají. Moduly, na nichž jiné moduly závisejí, musejí na konci své tabulky symbolů udržovat seznam odkazů, na nějž se ukazuje ze struktury `module`. Na obrázku 12.1 je vidět, že modul souborového systému `vfat` závisí na modulu souborového systému `fat`. Modul `fat` tedy obsahuje odkaz na modul `vfat`, tento odkaz byl přidán, když se nahrál modul `vfat`. Jádro volá inicializační rutinu modulu a pokud rutina proběhne úspěšně, je modul považován za nahraný. Adresa ukončovací rutiny modulu se umístí do datové struktury `module` modulu a tato rutina bude volána, až jádro bude modul odstraňovat. Nakonec se stav modulu změní na `RUNNING`.

12.2 Rušení modulu

Moduly je možno odstranit pomocí příkazu *rmmod*, vynuceně nahrané moduly ale démon `kerneld` odstraňuje automaticky, jakmile nejsou zapotřebí. Vždy, když vyprší časovač démona `kerneld`, provede démon systémové volání, jímž se požaduje odstranění všech nepoužívaných vynuceně nahraných modulů ze systému. Hodnota časovače se nastavuje při spuštění démona `kerneld`, v mém systému se kontrola provádí každých 180 sekund. Pokud například připojíme souborový systém `iso9660` a tento systém máme implementován jako modul, pak po odpojení mechaniky CD-ROM dojde v krátké době k odstranění modulu `iso9600` z jádra.

Modul není možno odstranit, dokud na něm závisejí nějaké komponenty jádra. Nemůžete například odstranit modul `vfat`, pokud máte připojen jeden nebo více souborových systémů `vfat`. Když se podíváte na výpis příkazu `lsmod`, uvidíte, že každý modul má své počitadlo:

```
Module:    #pages:   Used by:
msdos            5                 1
vfat             4                 1  (autoclean)
fat              6    [vfat msdos]  2  (autoclean)
```

Počitadlo představuje počet entit jádra, které na modulu závisejí. V předchozím příkladu oba moduly `vfat` a `msdos` závisejí na modulu `fat`, který má tedy počitadlo rovno dvěma. Moduly `vfat` a `msdos` jsou každý používány jednou, a to připojeným souborovým systémem. Pokud připojím další souborový systém `vfat`, počitadlo modulu `vfat` bude 2. Počitadlo modulu je uloženo v prvním dvouslově jeho obrazu.

Toto pole je lehce přetíženo, protože kromě počitadla obsahuje také příznaky `AUTOCLEAN` a `VISITED`. Oba tyto příznaky se používají u vynuceně nahraných modulů. Vynuceně nahrané moduly jsou označeny příznakem `AUTOCLEAN`, aby systém věděl, že mají být automaticky zrušeny. Příznak `VISITED` označuje moduly, které jsou používány jednou nebo

více systémovými komponentami, nastavuje se vždy, když nějaká komponenta modul používá. Vždy když systém požaduje po démonu `kerneld` odstranění nepotřebných vynuceně nahraných modulů, prohlédnou se všechny moduly v systému a hledají se kandidáti na zrušení. Prohlížejí se pouze moduly označené jako `AUTOCLEAN` ve stavu `RUNNING`. Pokud modul nemá nastaven příznak `VISITED`, odstraní se. Je-li příznak nastaven, vynuluje se a pokračuje se dalším modulem v systému.

7 Pokud se modul bude rušit, volá se jeho ukončovací rutina, která zajistí uvolnění prostředků jádra, alokovaných modulem.

Datová struktura `module` daného modulu se označí jako `DELETED` a vyřadí se ze seznamu modulů jádra. Všem modulům, na nichž rušený modul závisel, se upraví odkazy tak, aby věděl, že modul už je nepotřebuje. Nakonec se uvolní paměť jádra, přidělená modulu.

Odkazy na zdrojové texty jádra

1 – `kerneld is in the modules package along with insmod, Ismod and rmmod.`

2 – `Viz includ/-linux/kerneld.h`

3 – `Viz sys_get_kernel_syms() in kernel/module.c`

4 – `Viz include/linux/module.h`

5 – `Viz sys_create_module() in kernel/module.c.`

6 – `Viz sys_init_module() in kernel/module.c.`

7 – `Viz sys_delete_module() in kernel/module.c`

Kapitola

13

Zdrojový kód Linuxu

V této kapitole vysvětlujeme, kde ve zdrojovém kódu Linuxu byste měli hledat různé funkce jádra.

Ke čtení této knihy nepotřebujete znát programovací jazyk C, nepotřebujete ani zdrojový kód Linuxu, abyste pochopili, jak jádro funguje. Studium kódu jádra je pouze cvičení, díky němuž získáte hlubší pochopení chování operačního systému Linux. Tato kapitola představuje přehled zdrojového kódu jádra, jak je uspořádán a kde byste měli začít hledat určitou funkci.

13.1 Jak získat zdrojový kód jádra

Všechny hlavní distribuce Linuxu (Debian, Slackware, Red Hat a další) zdrojový kód přímo obsahují. Obvykle se operační systém nainstalovaný na vašem počítači vytváří přímo z těchto zdrojových kódů. Ze své podstaty ovšem zdrojové kódy nebývají úplně aktuální, a proto možná budete chtít získat nejnovější zdrojové kódy na některé z adres, uvedených v příloze C. Zdrojové kódy jsou uloženy na adrese `ftp://ftp.cs.helsinki.fi`* a ostatní zdroje jsou pouze zrcadla tohoto serveru. Helsinky jsou tedy vždy nejaktuálnější, ovšem servery jako MIT a Sunsite nejsou nikdy příliš pozadu.

Pokud nemáte přístup k webu, existuje řada prodejců, kteří na CD-ROM nabízejí archívy hlavních světových serverů za velmi přijatelné ceny. Někteří dokonce nabízejí předplatné na čtvrtletní nebo dokonce měsíční aktualizace.

---

* Poznámka korektora: V současnosti je primárním zdrojem linuxových zdrojových textů server *ftp://ftp.kernel.org*. V České republice je zrcadlo např. na *ftp://ftp.fi.muni.cz/pub/linux/kernel*.

Zdrojový kód jádra Linuxu používá velmi jednoduchý systém číslování. Každé sudé číslo jádra (například `2.0.30`) je stabilní, oficiálně distribuované jádro, každé liché číslo (například `2.1.42`) je vývojová verze jádra. Tato kniha vychází ze stabilní verze `2.0.30`. Vývojové verze jádra obsahují ty nejnovější funkce a podporují ta nejnovější zařízení. I když mohou být nestabilní, což vám nemusí vyhovovat, je velmi důležité, aby uživatelé Linuxu zkoušeli nejnovější verze. Díky tomu dochází k jejich širokému testování. Pamatujte si ale, že je velmi rozumné vždy provést zálohu stávajícího systému předtím, než začnete zkoušet nejnovější vývojovou verzi.

Změny zdrojových kódů jádra se distrubuují jako záplaty (patche). Utilita patch provádí úpravy ve zdrojových souborech. Pokud například používáte zdrojové soubory 2.0.29 a chcete přejít na verzi 2.0.30, měli byste si pořídit patch 2.0.30 a na stávající zdrojový kód použít tuto utilitu:

```
$ cd /usr/src/linux
$ patch -p1 < patch-2.0.30
```

Tím se ušetří kopírování celých zdrojových souborů, například přes pomalá modemová spojení. Dobrým zdrojem patchů jádra (oficiálních i neoficiálních) je webová adresa `http://www.linuxhq.com`.

13.2 Členění zdrojového kódu

Na nejvyšší úrovni zdrojového stromu `/usr/src/linux` uvidíte řadu adresářů:

arch Podadresář `arch` obsahuje veškerý na architektuře závislý kód jádra. Obsahuje další podadresáře, jeden pro každou podporovanou architekturu, například `i386` a `alpha`.

include Podadresář `include` obsahuje většinu hlavičkových souborů potřebných pro sestavení jádra. I on obsahuje další podadresáře členěné podle podporovaných architektur. Podadresář `include/asm` je odkaz na skutečný adresář potřebný pro danou architekturu, například `include/asm/i386`. Architekturu změníte tak, že upravíte soubor Makefile jádra a znovu spustíte konfigurační program jádra.

init Tento adresář obsahuje inicializační kód jádra a je to dobré místo, odkud začít studovat, jak jádro funguje.

mm Tento adresář obsahuje veškerý kód správy paměti. Kód správy paměti závislý na architektuře je uložen v adresáři `arch/*/mm`, např. `arch/i386/mm-/fault.c`.

drivers Adresář obsahuje všechny ovladače zařízení. Je dále členěn podle tříd ovladačů zařízení, například `block`.

ipc Adresář obsahuje kód pro meziprocesovou komunikaci.

fs Kód souborového systému. Je dále členěn na podadresáře pro jednotlivé podporované souborové systémy, například `vfat` a `ext2`.

kernel Hlavní kód jádra. Na architektuře závislé části jsou opět uloženy v `arch/*/kernel`.

net Síťový kód.

lib Tento adresář obsahuje kód knihoven jádra. Na architektuře závislé části kódu jsou uloženy v `arch/*/lib`.

scripts Tento adresář obsahuje skripty (například skripty `awk` a `tk`), používané při konfiguraci jádra.

13.3 Kde začít hledat

Představa hledání něčeho v tak velkém a složitém programu jako je jádro Linuxu může zastrašit. Působí jako obrovské klubko provázku bez konce a začátku. Prohlížení jedné části jádra často vede k hledání v dalších souvisejících částech a zanedlouho zapomenete, co jste vlastně hledali. V následujících částech jsou uvedena doporučení, kde ve struktuře zdrojových souborů co hledat.

13.3.1 Spouštění a inicializace systému

Na systémech s procesorem Intel se jádro spouští, když program `loadlin.exe` nebo *LILO* nahraje jádro do paměti a předá mu řízení. Tuto část najdete v `arch/i386/kernel/head.S`. Soubor `head.S` provede nějaké systémově specifické nastavení a poté skáče do rutiny `main()` v `init/main.c`.

13.3.2 Správa paměti

Kód je převážně soustředěn v `mm`, na architektuře závislé části najdete v `arch/*/mm`. Kód obsluhy výpadku stránky je v `mm/memory.c`, paměťové mapování a vyrovnávací paměť stránek je v `mm/filemap.c`. Vyrovnávací paměť bufferů je implementována v `mm/buffer.c` a odkládací vyrovnávací paměť v `mm/swap_state.c` a `mm/swapfile.c`.

13.3.3 Jádro

Většina důležitého obecného kódu jádra je v `kernel` se systémově závislými částmi v `arch/*/kernel`. Plánovač najdete v `kernel/sched.c`, kód rutiny fork v `kernel/fork.c`. Bottom half obsluha je implementována v `include/linux/interrupt.h`. Datová struktura `task_struct` se nachází v `include/linux/sched.h`.

13.3.4 PCI

Pseudoovladač zařízení PCI je v `drivers/pci/pci.c`, celosystémově platné definice v `include/linux/pci.h`. Každá architektura má nějaký specifický kód PCI BIOS, pro Alphu je v `arch/alpha/kernel/bios32.c`.

13.3.5 Meziprocesová komunikace

Vše je v `ipc`. Všechny IPC objekty Systemu V používají datovou strukturu `ipc_perm`, která je v `include/linux/ipc.h`. Zprávy Systemu V jsou implementovány v `ipc/msg.c`, semafory v `ipc/sem.c`. Roury jsou implementovány v `ipc/pipe.c`.

13.3.6 Obsluha přerušení

Obsluha přerušení je z největší části závislá na procesoru a architektuře systému. Obslužný kód přerušení pro Intel je v `arch/i386/kernel/irq.c`, definice v `include/asm/i386/irq.h`.

13.3.7 Ovladače zařízení

Největší část zdrojového kódu Linuxu tvoří jeho ovladače zařízení. Všechny ovladače zařízení jsou v adresáři `drivers`, který se ale dále dělí podle typu zařízení:

/block Ovladače blokových zařízení jako je třeba IDE (v `ide.c`). Pokud vás zajímá, jak se provádí obecná inicializace všech zařízení, která mohou obsahovat souborový systém, pak se podívejte na rutinu `device_setup()` v `drivers/block/genhd.c`. Provádí inicializaci nejen pevných disků, ale i například sítě, kterou potřebujete při připojení souborových systémů `nfs`. Bloková zařízení zahrnují jak zařízení IDE, tak zařízení SCSI.

/char Znaková zařízení jako `tty`, sériové porty a myš.

/cdrom Veškerý kód pro podporu CD-ROM. Zde najdete ovladače pro jednotlivá zařízení CD-ROM (například Soundblaster CDROM). Ovladač IDE CD je v `ide-cd.c` v adresáři `drivers/block`, ovladač SCSI CD je v `scsi.c` v adresáři `drivers/scsi`.

/pci Zdrojové kódy ovladače pseudozařízení PCI. Zde se dá zjistit, jak se subsystém PCI mapuje a inicializuje. PCI fixup kód pro architekturu Alpha AXP najdete v `/arch/alpha/kernel/bios32.c`.

/scsi Zde naleznete kód SCSI a ovladače všech zařízení SCSI, podporovaných Linuxem.

/net Zde najdete ovladače síťových zařízení, například ovladač ethernetové karty DECChip 21040 PCI je v `tulip.c`.

/sound Zde jsou ovladače zvukových karet.

13.3.8 Souborové systémy

Zdrojové kódy souborového systému `ext2` jsou v `fs/ext2` s definicemi datových struktur v `include/linux/ext2_fs.h`, `ext2_fs_i.h` a `ext2_fs_sb.h`. Datové struktury virtuálního souborového systému jsou popsány v `include/linux/fs.h`, kód je v `fs/*`. Vyrovnávací paměť bufferů je implementována ve `fs/buffer.c` včetně démona `update`.

13.3.9 Sítě

Kód podpory sítí je uložen v adresáři `net`, většina hlavičkových souborů v `include/net`. Kód soketů BSD je v `net/socket.c`, soketový kód pro sokety IPV4 je v `net/ipv4/af_inet.c`. Obecný kód pro podporu protokolů (včetně rutin pro obsluhu bufferů `sk_buff`) je v `net/core`, kód TCP/IP v `net/ipv4`. Ovladače síťových zařízení jsou v `drivers/net`.

13.3.10 Moduly

Kód pro podporu modulů je zčásti v jádře a zčásti v adresáři `modules`. Modulová část kódu jádra je v `kernel/modules.c`, datové struktury a démon `kerneld` v `include/linux/module.h` a `include/linux/kerneld.h`. Struktura objektového souboru `ELF` je popsána v `include/linux/elf.h`.

Dodatek

A

Datové struktury Linuxu

V této příloze je uveden seznam hlavních datových struktur používaných Linuxem, které jsme v této knize popisovali. Pro potřeby tisku byly jejich definice lehce upraveny.

block_dev_struct

Datová struktura block_dev_struct slouží k registraci blokových zařízení pro potřeby vyrovnávací paměti bufferů. Tyto struktury jsou uloženy ve vektoru blk_dev.


```
struct blk_dev_struct {
  void (*request_fn)(void);
  struct request * current_request;
  struct request   plug;
  struct tq_struct plug_tq;
};
```


buffer_head

Datová struktura buffer_head obsahuje informace o blokových bufferech ve vyrovnávací paměti bufferů.


```
/* stavové bity */
#define BH_Uptodate   0  /* 1 pokud buffer obsahuje platná data     */
#define BH_Dirty      1  /* 1 pokud je buffer modifikován           */
#define BH_Lock       2  /* 1 pokud je buffer zamčen                */
#define BH_Req        3  /* 0 je-li buffer zneplatněn               */
#define BH_Touched    4  /* 1 pracovalo-li se s bufferem (stárnutí) */
```

```
#define BH_Has_aged   5  /* 1 pokud buffer zestárnul             */
#define BH_Protected  6  /* 1 je-li buffer chráněn               */
#define BH_FreeOnIO   7  /* 1 je-li nutné po IO zrušit buffer_head  */

struct buffer_head {
  /* První řádek cache: */
  unsigned long     b_blocknr;   /* číslo bloku                  */
  kdev_t            b_dev;       /* zařízení (B_FREE = volný)    */
  kdev_t            b_rdev;      /* skutečné zařízení            */
  unsigned long     b_rsector;   /* skutečné umístění na disku   */
  struct buffer_head *b_next;    /* seznam hashovací fronty      */
  struct buffer_head *b_this_page;     /* cyklický seznam bufferů
                                          v jedné stránce        */

  /* Druhý řádek cache: */
  unsigned long     b_state;     /* stav bufferu (viz výše)      */
  struct buffer_head  *b_next_free;
  unsigned int      b_count;     /* uživatelé tohoto bloku       */
  unsigned long     b_size;      /* velikost bloku               */

  /* následují pro výkon nekritická data */
  char              *b_data;     /* ukazatel na datový blok      */
  unsigned int      b_list;      /* seznam, v němž je tento buffer*/
  unsigned long     b_flushtime; /* čas, kdy by se měl buffer zapsat */
  unsigned long     b_lru_time;  /* čas posledního použití bufferu */
  struct wait_queue  *b_wait;
  struct buffer_head *b_prev;    /* obousměrný hashovací seznam  */
  struct buffer_head *b_prev_free; /* obousměrný seznam bufferů  */
  struct buffer_head *b_reqnext; /* fronta požadavků             */
};
```

device

Každé síťové zařízení v systému je reprezentováno datovou strukturou `device`.


```
struct device
{

 /*
 * Toto je první “viditelná” položka této struktury (tedy část, kterou
 * uživatel vidí v souboru “Space.c”). .c" file). Je to jméno rozhraní.
 */
 char                     *name;

  /* V/V specifické údaje */
  unsigned long           rmem_end;     /* "recv" konec sd. pam.   */
  unsigned long           rmem_start;   /* "recv" zač. sd. pam.    */
  unsigned long           mem_end;      /* konec sdíl. paměti      */
  unsigned long           mem_start;    /* zač. sdílené paměti     */
  unsigned long           base_addr;    /* V/V adresa zařízení     */
  unsigned char           irq;          /* číslo přerušení         */

  /* Nízkoúrovňové stavové příznaky */
  volatile unsigned char  start,        /* začátek operace         */
                          interrupt;    /* došlo přerušení         */
  unsigned long           tbusy;        /* vysílač obsazen         */
  struct device           *next;

  /* Inicializační funkce zařízení. Volá se jen jednou.            */
  int                      (*init)(struct device *dev);

  /* Následující položky potřebují některá zařízení, nejsou ale
     obvykle uváděny ve Space.c. */
  unsigned char           if_port;      /* volitelné AUI,TP        */
  unsigned char           dma;          /* DMA kanál               */

  struct enet_statistics* (*get_stats)(struct device *dev);
```

```
/*
 * Konec “viditelné části struktury. Všechny následující položky
 * jsou interní systémové položky a mohou se libovolně měnit
 */

/* Pro potřeby budoucího síťového kódu                          */
unsigned long           trans_start; /* čas (jiffies) posledního
                                        vysílání                */
unsigned long           last_rx;     /* čas posl. příjmu        */
unsigned short          flags;       /* příznaky rozhr. (BSD)   */
unsigned short          family;      /* ID adresové rodiny      */
unsigned short          metric;      /* směrovací metrika       */
unsigned short          mtu;         /* MTU                     */
unsigned short          type;        /* hardwarový typ          */
unsigned short          hard_header_len; /* délka HW hlavičky   */
void                    *priv;       /* privátní data           */

/* Adresové informace */
unsigned char           broadcast[MAX_ADDR_LEN];
unsigned char           pad;
unsigned char           dev_addr[MAX_ADDR_LEN];
unsigned char           addr_len;    /* délka hw adresy         */
unsigned long           pa_addr;     /* protokolová adresa      */
unsigned long           pa_brdaddr;  /* protokolová vysílací a. */
unsigned long           pa_dstaddr   /* další protokolová adr.  */
unsigned long           pa_mask;     /* síťová maska            */
unsigned short          pa_alen;     /* délka protok. adresy    */

struct dev_mc_list      *mc_list;    /* hromadné adresy         */
int                     mc_count;    /* kolik hrom. adres       */

struct ip_mc_list       *ip_mc_list; /* hrom. filtrační řetězec */
__u32                   tx_queue_len;/* max rámců ve frontě     */

/* Kvůli vyvažování zátěže mezi páry zařízení */
unsigned long           pkt_queue;   /* paketů ve frontě        */
struct device           *slave;      /* druhé zařízení          */
struct net_alias_info   *alias_info; /* hlavní alias zařízení   */
struct net_alias        *my_alias;   /* další aliasy            */
```

```
  /* Ukazatel na buffery rozhraní */
  struct sk_buff_head     buffs[DEV_NUMBUFFS];

  /* Ukazatele na služební rutiny rozhraní */
  int                     (*open)(struct device *dev);
  int                     (*stop)(struct device *dev);
  int                     (*hard_start_xmit) (struct sk_buff *skb,
                            struct device *dev);
  int                     (*hard_header)      (struct sk_buff *skb,
                                               struct device *dev,
                                               unsigned short type,
                                               void *daddr,
                                               void *saddr,
                                               unsigned len);
  int                     (*rebuild_header)(  void *eth,
                                               struct device *dev,
                                               unsigned long raddr,
                                               struct sk_buff *skb);
  void                    (*set_multicast_list)(struct device *dev);
  int                     (*set_mac_address)  (struct device *dev,
                                               void *addr);
  int                     (*do_ioctl)(struct device *dev,
                                               struct ifreq *ifr,
                                               int cmd);
  int                     (*set_config)(struct device *dev,
                                               struct ifmap *map);
  void                    (*header_cache_bind)(struct hh_cache **hhp,
                                               struct device *dev,
                                               unsigned short htype,
                                               __u32 daddr);
  void                    (*header_cache_update)(struct hh_cache *hh,
                                               struct device *dev,
                                              unsigned char *  haddr);
  int                     (*change_mtu)(struct device *dev,
                                               int new_mtu);
  struct iw_statistics*   (*get_wireless_stats)(struct device *dev);
};
```

Průvodce jádrem operačního systému Linux

device_struct

Datová struktura `device_struct` slouží k registraci znakových a blokových zařízení (obsahuje jejich jméno a množinu souborových operací, které zařízení podporuje). Každý platný prvek vektorů `chrdevs` a `blkdevs` reprezentuje jedno znakové, respektive blokové zařízení.

4

```
struct device_struct {
    const char * name;
    struct file_operations * fops;
};
```

file

5

Datovou strukturou `file` je reprezentován každý otevřený soubor, soket a podobně.

```
struct file {
  mode_t f_mode;
  loff_t f_pos;
  unsigned short f_flags;
  unsigned short f_count;
  unsigned long f_reada, f_ramax, f_raend, f_ralen, f_rawin;
  struct file *f_next, *f_prev;
  int f_owner;          /* pid nebo -pgrp, kam se má poslat SIGIO
*/
  struct inode * f_inode;
  struct file_operations * f_op;
  unsigned long f_version;
  void *private_data;  /* nutné pro ovladače tty a některé další */
};
```

6

files_struct

Datová struktura `files_struct` popisuje soubory, které má proces otevřen.

```
struct files_struct {
  int count;
  fd_set close_on_exec;
  fd_set open_fds;
  struct file * fd[NR_OPEN];
};
```

fs_struct

7

```
struct fs_struct {
  int count;
  unsigned short umask;
  struct inode * root, * pwd;
};
```

gendisk

Datová struktra `gendisk` obsahuje informace o pevném disku. Používá se v době inicializace, kdy se naleznou pevné disky a pak se na nich hledají oblasti.

8

```
struct hd_struct {
    long start_sect;
    long nr_sects;
};

struct gendisk {
    int major;                    /* hlavní číslo ovladače */
    const char *major_name;       /* jméno hlavního ovladače */
    int minor_shift;              /* kolikrát se posouvá vedlejší
                                     číslo než vznikne skutečné
                                     vedlejší číslo */
    int max_p;                    /* maximum oblastí na zařízení */
    int max_nr;                   /* maximální počet zařízení */

    void (*init)(struct gendisk *);
                                  /* inicializace volaná než začneme
                                    dělat naše věci */
    struct hd_struct *part;       /* tabulka oblastí */
    int *sizes;                   /* velikost zařízení v blocích,
                                     kopíruje se do blk_size[] */
    int nr_real;                  /* počet reálných zařízení */

    void *real_devices;           /* pro interní použití */
    struct gendisk *next;
};
```

inode

9

Datová struktura VFS `inode` obsahuje informace o souboru nebo adresáři na disku.

```
struct inode {
    kdev_t                              i_dev;
    unsigned long                       i_ino;
    umode_t                             i_mode;
    nlink_t                             i_nlink;
    uid_t                               i_uid;
    gid_t                               i_gid;
    kdev_t                              i_rdev;
    off_t                               i_size;
    time_t                              i_atime;
    time_t                              i_mtime;
    time_t                              i_ctime;
    unsigned long                       i_blksize;
    unsigned long                       i_blocks;
    unsigned long                       i_version;
    unsigned long                       i_nrpages;
    struct semaphore                    i_sem;
    struct inode_operations             *i_op;
    struct super_block                  *i_sb;
    struct wait_queue                   *i_wait;
    struct file_lock                    *i_flock;
    struct vm_area_struct               *i_mmap;
    struct page                         *i_pages;
    struct dquot                        *i_dquot[MAXQUOTAS];
    struct inode                        *i_next, *i_prev;
    struct inode                        *i_hash_next, *i_hash_prev;
    struct inode                        *i_bound_to, *i_bound_by;
    struct inode                        *i_mount;
    unsigned short                      i_count;
    unsigned short                      i_flags;
    unsigned char                       i_lock;
    unsigned char                       i_dirt;
    unsigned char                       i_pipe;
    unsigned char                       i_sock;
    unsigned char                       i_seek;
```

```
    unsigned char                       i_update;
    unsigned short                      i_writecount;
    union {
        struct pipe_inode_info          pipe_i;
        struct minix_inode_info         minix_i;
        struct ext_inode_info           ext_i;
        struct ext2_inode_info          ext2_i;
        struct hpfs_inode_info          hpfs_i;
        struct msdos_inode_info         msdos_i;
        struct umsdos_inode_info        umsdos_i;
        struct iso_inode_info           isofs_i;
        struct nfs_inode_info           nfs_i;
        struct xiafs_inode_info         xiafs_i;
        struct sysv_inode_info          sysv_i;
        struct affs_inode_info          affs_i;
        struct ufs_inode_info           ufs_i;
        struct socket                   socket_i;
        void                            *generic_ip;
    } u;
};
```

ipc_perm

10 Datová struktura `ipc_perm` popisuje přístupová práva k meziprocesovým komunikačním objektům Systemu V.

```
struct ipc_perm
{
  key_t  key;
  ushort uid;       /* euid a egid vlastníka */
  ushort gid;
  ushort cuid;      /* euid a egid autora */
  ushort cgid;
  ushort mode;      /* přístupové režimy viz příznaky režimů níže */
  ushort seq;       /* sekvenční číslo */
};
```

irqaction

11 Datová struktura `irqaction` slouží k popisu handlerů přerušení v systému.

```
struct irqaction {
  void (*handler)(int, void *, struct pt_regs *);
  unsigned long flags;
  unsigned long mask;
  const char *name;
  void *dev_id;
  struct irqaction *next;
};
```

linux_binfmt

12 Každý binární souborový formát, jemuž Linux rozumí, je reprezentován datovou strukturou `linux_binfmt`.

```
struct linux_binfmt {
  struct linux_binfmt * next;
  long *use_count;
  int (*load_binary)(struct linux_binprm *, struct  pt_regs *
regs);
  int (*load_shlib)(int fd);
  int (*core_dump)(long signr, struct pt_regs * regs);
};
```

mem_map_t

13 Datová struktura `mem_map_t` (označovaná také jako stránka) obsahuje informace o každé stránce fyzické paměti.

```
typedef struct page {
  /* tyto údaje musejí být první (kvůli manipulaci s volnou pamětí) */
  struct page          *next;
  struct page          *prev;
  struct inode         *inode;
  unsigned long        offset;
  struct page          *next_hash;
  atomic_t             count;
```

```
  unsigned                flags;      /* atomické příznaky, mohou se
                                         měnit asynchronně */
  unsigned                dirty:16,
                          age:8;
  struct wait_queue       *wait;
  struct page             *prev_hash;
  struct buffer_head      *buffers;
  unsigned long           swap_unlock_entry;
  unsigned long           map_nr; /* page->map_nr == page - mem_map */
} mem_map_t;
```

mm_struct

14 Datová struktura mm_struct slouží k popisu virtuální paměti procesů.

```
struct mm_struct {
  int count;
  pgd_t * pgd;
  unsigned long context;
  unsigned long start_code, end_code, start_data, end_data;
  unsigned long start_brk, brk, start_stack, start_mmap;
  unsigned long arg_start, arg_end, env_start, env_end;
  unsigned long rss, total_vm, locked_vm;
  unsigned long def_flags;
  struct vm_area_struct * mmap;
  struct vm_area_struct * mmap_avl;
  struct semaphore mmap_sem;
};
```

pci_bus

15 Každá sběrnice PCI v systému je reprezentována strukturou pci_bus.

```
struct pci_bus {
  struct pci_bus  *parent;      /* rodičovská sběrnice můstku */
  struct pci_bus  *children;    /* seznam můstků na této sběrnici */
  struct pci_bus  *next;        /* seznam všech sběrnic */

  struct pci_dev  *self;        /* můstek z pohledu rodiče */
```

```
  struct pci_dev  *devices;      /* zařízení pod můstkem */

  void    *sysdata;              /* odkaz na systémově závislá data */

  unsigned char  number;         /* číslo sběrnice */
  unsigned char  primary;        /* číslo primárního můstku */
  unsigned char  secondary;      /* číslo sekundárního můstku */
  unsigned char  subordinate;    /* číslo podřízené sběrnice */
};
```

pci_dev

16 Každé zařízení PCI v systému včetně můstků PCI-PCI a PCI-ISA je reprezentováno datovou strukturou `pci_dev`.

```
/*
 * Existuje jedna struktura pci_dev pro každou dvojici číslo slotu/
 * číslo funkce:
 */
struct pci_dev {
  struct pci_bus  *bus;        /* číslo sběrnice, na níž je toto zař.
*/
  struct pci_dev  *sibling;    /* další zařízení na této sběrnici */
  struct pci_dev  *next;       /* seznam všech zařízení */

  void    *sysdata;            /* odkaz na systémově specifická data
*/

  unsigned int  devfn;         /* kódovaný index zařízení a funkce */
  unsigned short  vendor;
  unsigned short  device;
  unsigned int  class;         /* 3 bajty: (base, sub, prog-if) */
  unsigned int  master : 1;    /* nastaveno, jde-li o zař. master */
  /*
   * Teoreticky je možno úroveň přerušení přečíst v konfiguraci
   * a všechno by fungovalo. Bohužel starší čipy PCI nepodporují
   * tyto registry a namísto toho vracejí nulu. Například řadič
   * Vision864-P rev 0 umí používat INTA, vrací ale 0 v registrech
   * přerušovací linky a pinu. Funkce pci_init() inicializuje
```

```
   * tuto položku hodnotou v PCI_INTERRUPT_LINE a pokud to bude
   * nutné, mění ji funkce pcibios_fixup(). Údaj nesmí být nulový,
   * ledaže zařízení vůbec negeneruje přerušení.
*/
  unsigned char  irq;     /* přerušení generované zařízením */
};
```

request

17 Datová struktura request se používá k vytváření požadavků na bloková zařízení v systému. Požadavky vždy znamenají zápis nebo čtení bloku dat do nebo z vyrovnávací paměti bufferů.

```
struct request {
    volatile int rq_status;
#define RQ_INACTIVE              (-1)
#define RQ_ACTIVE                1
#define RQ_SCSI_BUSY             0xffff
#define RQ_SCSI_DONE             0xfffe
#define RQ_SCSI_DISCONNECTING    0xffe0

    kdev_t rq_dev;
    int cmd;        /* READ nebo WRITE */
    int errors;
    unsigned long sector;
    unsigned long nr_sectors;
    unsigned long current_nr_sectors;
    char * buffer;
    struct semaphore * sem;
    struct buffer_head * bh;
    struct buffer_head * bhtail;
    struct request * next;
};
```

rtable

18 Každá datová struktura `rtable` obsahuje informace o trase používané k odesílání IP paketů. Tato struktura se používá ve vyrovnávací paměti tras.

```
struct rtable
{
    struct rtable      *rt_next;
    __u32               rt_dst;
    __u32               rt_src;
    __u32               rt_gateway;
    atomic_t            rt_refcnt;
    atomic_t            rt_use;
    unsigned long      rt_window;
    atomic_t            rt_lastuse;
    struct hh_cache    *rt_hh;
    struct device      *rt_dev;
    unsigned short     rt_flags;
    unsigned short     rt_mtu;
    unsigned short     rt_irtt;
    unsigned char      rt_tos;
};
```

semaphore

19 Semafory slouží k ochraně kritických datových struktur a oblastí kódu.

```
struct semaphore {
    int count;
    int waking;
    int lock ;                  /* aby bylo testování atomické */
    struct wait_queue *wait;
};
```

sk_buff

20 Datová struktura `sk_buff` slouží k popisu síťových dat při jejich předávání mezi protokolovými vrstvami.

```
struct sk_buff
{
  struct sk_buff    *next;        /* další buffer v seznamu        */
  struct sk_buff    *prev;        /* předchozí buffer v seznamu    */
  struct sk_buff_head *list;      /* seznam, v němž jsme           */
  int               magic_debug_cookie;
  struct sk_buff    *link3;       /* odkaz na buffery IP protokolu */
  struct sock       *sk;          /* soket, který nás vlastní      */
  unsigned long     when;         /* používáno k výpočtu rtt       */
  struct timeval    stamp         /* čas, kdy jsme dorazili        */
  struct device     *dev;         /* zařízení, na němž jsme        */
                                  dorazili, z nějž budeme odesláni */
  union
  {
      struct tcphdr       *th;
      struct ethhdr       *eth;
      struct iphdr        *iph;
      struct udphdr       *uh;
      unsigned char       *raw;
      /* pro předávání handlů souborů v soketech domény Unix */
      void                *filp;
  } h;

  union
  {
      /* Část informací fyzické vrstvy */
      unsigned char       *raw;
      struct ethhdr       *ethernet;
  } mac;

  struct iphdr            *ip_hdr;      /* pro IPPROTO_RAW         */
  unsigned long           len;          /* skutečná délka dat      */
  unsigned long        csum;            /* kontrolní součet        */
  __u32                 saddr;          /* zdrojová IP adresa      */
```

```
  __u32            daddr;        /* cílová IP adresa             */
  __u32            raddr;        /* IP adresa dalšího skoku      */
  __u32            seq;          /* sekvenční číslo (TCP)        */
  __u32            end_seq;      /* seq [+ fin] [+ syn] + datalen */
  __u32            ack_seq;      /* potvrzovací sekvenční číslo TCP)
*/
  unsigned char    proto_priv[16];
  volatile char     acked,       /* jsme potvrzeni?              */
                   used,         /* jsme používáni?              */
                   free,         /* jak tento buffer uvolnit     */
                   arp;          /* dokončen překlad IP/ARP      */
  unsigned char    tries,        /* kolikrát zkoušeno            */
                   lock,         /* jsme zamčení?                */
                   localroute,   /* lokální směrování tohoto rámce */
                   pkt_type,     /* třída paketu                 */
                   pkt_bridged,  /* trasa pro můstky             */
                   ip_summed;    /* kontrolní součet IP          */
#define PACKET_HOST        0     /* nám                          */
#define PACKET_BROADCAST   1     /* všem                         */
#define PACKET_MULTICAST   2     /* skupině                      */
#define PACKET_OTHERHOST   3     /* někomu jinému                */
  unsigned short   users;        /* počitadlo uživatelů
                                  viz datagram.c,tcp.c           */
  unsigned short   protocol;     /* protokol                     */
  unsigned int     truesize;     /* velikost bufferu             */
  atomic_t         count;        /* referenční počitadlo         */
  struct sk_buff   *data_skb;    /* odkaz na datový buffer       */
  unsigned char    *head;        /* začátek bufferu              */
  unsigned char    *data;        /* začátek dat                  */
  unsigned char    *tail;        /* ukazatel na patičku          */
  unsigned char    *end;         /* ukazatel na konec            */
  void             (*destructor)(struct sk_buff *);
                                 /* funkce rušení                */
  __u16            redirport;    /* port pro přesměrování        */
};
```

sock

Každá datová struktura `sock` obsahuje protokolově specifické informace o BSD soketu. Například pro soket domény INET bude tato struktura obsahovat všechny informace specifické pro protokoly TCP/IP a UDP/IP.


```
struct sock
{
    /* Toto musí být první */
    struct sock           *sklist_next;
    struct sock           *sklist_prev;

    struct options        *opt;
    atomic_t              wmem_alloc;
    atomic_t              rmem_alloc;
    unsigned long         allocation;  /* režim alokace */
    __u32                 write_seq;
    __u32                 sent_seq;
    __u32                 acked_seq;
    __u32                 copied_seq;
    __u32                 rcv_ack_seq;
    unsigned short        rcv_ack_cnt; /* počitadlo potvrzení */
    __u32                 window_seq;
    __u32                 fin_seq;
    __u32                 urg_seq;
    __u32                 urg_data;
    __u32                 syn_seq;
    int                   users;       /* počitadlo uživatelů */
  /*
   *    Ne všechny údaje jsou sice typu "volatile",
   *    některé ale jsou, takže stejně dobře mohou být všechny.
   */
    volatile char         dead,
                          urginline,
                          intr,
                          blog,
                          done,
                          reuse,
                          keepopen,
```

```
                      linger,
                      delay_acks,
                      destroy,
                      ack_timed,
                      no_check,
                      zapped,
                      broadcast,
                      nonagle,
                      bsdism;
unsigned long         lingertime;
int                   proc;

struct sock           *next;
struct sock           **pprev;
struct sock           *bind_next;
struct sock           **bind_pprev;
struct sock           *pair;
int                   hashent;
struct sock           *prev;
struct sk_buff        *volatile send_head;
struct sk_buff        *volatile send_next;
struct sk_buff        *volatile send_tail;
struct sk_buff_head   back_log;
struct sk_buff        *partial;
struct timer_list     partial_timer;
long                  retransmits;
struct sk_buff_head   write_queue,
                      receive_queue;
struct proto          *prot;
struct wait_queue     **sleep;
__u32                 daddr;
__u32                 saddr;        /* Odesílatel */
__u32                 rcv_saddr;
unsigned short        max_unacked;
unsigned short        window;
__u32                 lastwin_seq;  /* sekv. číslo, kdy jsme
                                       naposledy aktualizovali
                                       nabízené okno */
```

```
    __u32                high_seq;    /* sekv. číslo, kdy jsme
                                         naposledy opakovali */
    volatile unsigned long  ato;      /* timeout potvrzení */
    volatile unsigned long lrcvtime;  /* jiffies při posledním
                                         příjmu dat */
    volatile unsigned long  idletime; /* jiffies při posl.příjmu */
    unsigned int         bytes_rcv;
/*
 *    mss je min(mtu, max_window)
 */
    unsigned short       mtu;         /* mss dohodnutné při sync */
    volatile unsigned short mss;      /* efektivní mss, mění se  */
    volatile unsigned short user_mss; /* mss požad. uživ. v ioctl*/
    volatile unsigned short max_window;
    unsigned long        window_clamp;
    unsigned int         ssthresh;
    unsigned short       num;
    volatile unsigned short cong_window;
    volatile unsigned short cong_count;
    volatile unsigned short packets_out;
    volatile unsigned short shutdown;
    volatile unsigned long  rtt;
    volatile unsigned long  mdev;
    volatile unsigned long  rto;

    volatile unsigned short backoff;
    int             err, err_soft;    /* pravidelně se objevující
                                         chyby, ne jen timeout */
    unsigned char           protocol;
    volatile unsigned char  state;
    unsigned char           ack_backlog;
    unsigned char           max_ack_backlog;
    unsigned char           priority;
    unsigned char           debug;
    int                     rcvbuf;
    int                     sndbuf;
    unsigned short          type;
    unsigned char   localroute;       /* směrovat jen lokálně */
```

```
/*
 *    Tady budou uloženy všechny specifické volitelné údaje.
 */
    union
    {
          struct unix_opt       af_unix;
#if defined(CONFIG_ATALK) || defined(CONFIG_ATALK_MODULE)
        struct atalk_sock       af_at;
#endif
#if defined(CONFIG_IPX) || defined(CONFIG_IPX_MODULE)
        struct ipx_opt          af_ipx;
#endif
#ifdef CONFIG_INET
        struct inet_packet_opt  af_packet;
#ifdef CONFIG_NUTCP
        struct tcp_opt          af_tcp;
#endif
#endif
    } protinfo;
/*
 *    'Privátní oblast' IP
 */
    int                   ip_ttl;             /* nastavení TTL */
    int                   ip_tos;             /* TOS */
    struct tcphdr         dummy_th;
    struct timer_list     keepalive_timer;    /* časovač
                                                 TCP keepalive   */
    struct timer_list     retransmit_timer;   /* časovač
                                                 TCP retransmit  */
    struct timer_list     delack_timer;       /* časovač potvrzovací
                                                 TCP             */
    int                   ip_xmit_timeout;    /* proč běží timeout */
    struct rtable         *ip_route_cache;    /* trasa v cache */
    unsigned char         ip_hdrincl;         /* včetně hlaviček? */
#ifdef CONFIG_IP_MULTICAST
    int                   ip_mc_ttl;          /* multicasting TTL */
    int                   ip_mc_loop;         /* loopback */
```

```
    char              ip_mc_name[MAX_ADDR_LEN];  /* jméno hrom.zař. */
    struct ip_mc_socklist   *ip_mc_list;      /* pole skupiny */
#endif

/*
 *     Tato část slouží funkcím timeoutu (timer.c).
 */
    int                    timeout;     /* Na co čekáme? */
    struct timer_list      timer; /* TIME_WAIT/receive časovač */
    struct timeval         stamp;
 /*
  *     Identd
  */
    struct socket          *socket;
  /*
   *     Callbacks
   */
    void                   (*state_change)(struct sock *sk);
    void                   (*data_ready)(struct sock *sk,int bytes);
    void                   (*write_space)(struct sock *sk);
    void                   (*error_report)(struct sock *sk);

};
```

socket

Každá datová struktura `socket` obsahuje informace o BSD soketu. Neexistuje nezávisle, je vždy součástí VFS inodu.

```
struct socket {
  short             type;         /* SOCK_STREAM, ...              */
  socket_state      state;
  long              flags;
  struct proto_ops    *ops;       /* protokoly dělají téměř vše    */
  void                *data;      /* protokolová data              */
  struct socket       *conn;      /* server, k němuž jsme připojeni */
  struct socket       *iconn;     /* nedokončené klientské spojení */
  struct socket       *next;
  struct wait_queue   **wait;     /* ukazatel na místo, kde čekáme */
```

```
  struct inode        *inode;
  struct fasync_struct *fasync_list;    /* asynchronní seznam
                                           čekatelů  */
  struct file         *file;            /* souborový ukazatel     */
};
```

task_struct

23 Každá datová struktura `task_struct` popisuje jeden proces v systému.

```
struct task_struct {
/* následující údaje neměnit */
  volatile long     state;       /* -1 unrunnable, 0 runnable,
                                    >0 pozastavený */
  long              counter;
  long              priority;
  unsigned          long signal;
  unsigned          long blocked;/* bitmapa maskovaných signálů */
  unsigned          long flags;  /* procesové příznaky, viz dále */
  int errno;
  long              debugreg[8]; /* hardwarové ladicí registry */
  struct exec_domain      *exec_domain;
/* různé údaje */
  struct linux_binfmt     *binfmt;
  struct task_struct      *next_task, *prev_task;
  struct task_struct      *next_run,  *prev_run;
  unsigned long           saved_kernel_stack;
  unsigned long           kernel_stack_page;
  int                     exit_code, exit_signal;
  /* ??? */
  unsigned long           personality;
  int                     dumpable:1;
  int                     did_exec:1;
  int                     pid;
  int                     pgrp;
  int                     tty_old_pgrp;
  int                     session;
  int                     leader;
```

```
  int                     groups[NGROUPS];
  /*
   * Ukazatele na (původní) rodičovský proces, nejmladšího syna,
   * nejmladšího a nejstaršího sourozence. (p->father je možno
   * nahradit p->p_pptr->pid)
   */
  struct task_struct      *p_opptr, *p_pptr, *p_cptr,
                            *p_ysptr, *p_osptr;
  struct wait_queue       *wait_chldexit;
  unsigned short          uid,euid,suid,fsuid;
  unsigned short          gid,egid,sgid,fsgid;
  unsigned long           timeout, policy, rt_priority;
  unsigned long           it_real_value, it_prof_value,
                             it_virt_value;
  unsigned long           it_real_incr, it_prof_incr, it_virt_incr;
  struct timer_list       real_timer;
  long                    utime, stime, cutime, cstime, start_time;
/* údaje pro obsluhu výpadku a odkládání */
  unsigned long           min_flt, maj_flt, nswap, cmin_flt,
                             cmaj_flt, cnswap;
  int swappable:1;
  unsigned long      swap_address;
  unsigned long      old_maj_flt;/* stará hodnota maj_flt */
  unsigned long      dec_flt;    /* počitadlo výpadků     */
  unsigned long      swap_cnt;   /* počet příště odkládaných str */
/* limity */
  struct rlimit      rlim[RLIM_NLIMITS];
  unsigned short     used_math;
  char              comm[16];
/* informace souborového systému */
  int                link_count;
  struct tty_struct  *tty;            /* NULL, není-li tty */
/* údaje meziprocesové komunikace */
  struct sem_undo    *semundo;
  struct sem_queue   *semsleeping;
/* ldt této úlohy, používá Wine. Pokud je NULL, použije se de-
fault_ldt. */
  struct desc_struct *ldt;
```

```
/* tss této úlohy */
  struct thread_struct tss;
/* informace souborového systému */
  struct fs_struct     *fs;
/* informace o otevřených souborech */
  struct files_struct  *files;
/* informace správy paměti */
  struct mm_struct     *mm;
/* obsluha signálů */
  struct signal_struct *sig;
#ifdef __SMP__
  int                   processor;
  int                   last_processor;
  int                   lock_depth;     /* hloubka zamčení */
#endif
};
```

timer_list

24 Datová struktura `timer_list` slouží k implementaci reálných časovačů procesů.

```
struct timer_list {
  struct timer_list *next;
  struct timer_list *prev;
  unsigned long expires;
  unsigned long data;
  void (*function)(unsigned long);
};
```

tq_struct

25 Každá fronta úloh `tq_struct` obsahuje informace o úkolech, které byly zařazeny do fronty. Jedná se obvykle o činnosti požadované ovladačem zařízení, které ale není nutné provést okamžitě.

```
struct tq_struct {
    struct tq_struct *next;   /* seznam prvků */
    int sync;                 /* inicializuje se na nulu */
```

```
    void (*routine)(void *);  /* funkce, která se má volat */
    void *data;               /* parametr funkce */
};
```


vm_area_struct

Každá datová struktura `vm_area_struct` popisuje oblast virtuální paměti procesu.

```
struct vm_area_struct {
  struct mm_struct * vm_mm;  /* parametry oblasti */
  unsigned long vm_start;
  unsigned long vm_end;
  pgprot_t vm_page_prot;
  unsigned short vm_flags;
/* AVL strom oblastí řazený podle adres */
  short vm_avl_height;
  struct vm_area_struct * vm_avl_left;
  struct vm_area_struct * vm_avl_right;
/* Seznam oblastí řazený podle adres */
  struct vm_area_struct * vm_next;
/* Pro oblasti s inody kruhový seznam inode->i_mmap */
/* Pro sdílené oblasti kruhový seznam uživatelů */
/* jinak nepoužito */
  struct vm_area_struct * vm_next_share;
  struct vm_area_struct * vm_prev_share;
/* další */
  struct vm_operations_struct * vm_ops;
  unsigned long vm_offset;
  struct inode * vm_inode;
  unsigned long vm_pte;      /* sdílená paměť */
};
```


Odkazy na zdrojové texty jádra

1 – Viz `include/linux/`
2 – Viz `include/linux/fs.h`
3 – Viz `include/linux/netdevice.h`

4 – `Viz fs/devices.c`
5 – `Viz include/linux/fs.h`
6 – `Viz include/linux/sched.h`
7 – `Viz include/linux/sched.h`
8 – `Viz include/linux/genhd.h`
9 – `Viz include/linux/fs.h`
10 – `Viz include/linux/ipc.h`
11 – `Viz include/linux/interrupt.h`
12 – `Viz include/linux/binfmts.h`
13 – `Viz include/linux/mm.h`
14 – `Viz include/linux/sched.h`
15 – `Viz include/linux/pci.h`
16 – `Viz include/linux/pci.h`
17 – `Viz include/linux/blkdev.h`
18 – `Viz include/linux/route.h`
19 – `Viz include/linux/semaphore.h`
20 – `Viz include/linux/skbuff.h`
21 – `Viz include/linux/net.h`
22 – `Viz include/linux/net.h`
23 – `Viz include/linux/sched.h`
24 – `Viz include/linux/timer.h`
25 – `Viz include/linux/tqueue.h`
26 – `Viz include/linux/mm.h`

Dodatek

B

Procesor Alpha AXP

Procesory Alpha AXP mají 64bitovou architekturu RISC load/store, navrženou s ohledem na rychlost. Všechny registry jsou dlouhé 64 bitů, procesor obsahuje 32 celočíselných registrů a 32 registrů v plovoucí řádové čárce. Celočíselný registr 31 a reálný registr 31 slouží pro nulové operace. Čtením z nich se získá nulová hodnota, zápis do nich nemá žádný efekt. Všechny operace jsou dlouhé 32 bitů a v paměti je možno provádět čtení nebo zápisy. Architektura umožňuje různé implementace do té doby, dokud implementace dodržují návrh architektury.

Neexistují instrukce pro přímou manipulaci s hodnotami v paměti, všechny datové manipulace se provádějí mezi registry. Pokud tedy budete chtít inkrementovat počitadlo v paměti, musíte je nejprve načíst do registru, modifikovat je a zapsat je zpět do paměti. Pro vzájemnou interakci slouží pouze jedna instrukce pro zápis do registru nebo paměti a jiná pro čtení registru nebo paměti. Zajímavou funkcí procesoru Alpha AXP je to, že některé instrukce mohou generovat příznaky, například při testování dvou registrů na shodu, ovšem výsledek se neuloží do stavového registru procesoru, ale do třetího registru. Na první pohled to může vypadat podivně, odstranění závislosti na stavových registrech ale znamená, že se usnadňuje konstrukce procesoru, který v jednom cyklu provádí více instrukcí. Instrukce s nezávislými registry nemusejí vzájemně čekat na své dokončení jak by tomu bylo, kdyby používaly společný stavový registr. Malý počet operací s pamětí a velký počet registrů rovněž usnadňuje paralelní zpracovávání instrukcí.

Architektura Alpha AXP používá skupinu subrutin, zvanou *privileged architecture library code* (PALcode). PALcode je specifický pro operační systém, implementaci architektury Alpha AXP v procesoru a hardware systému. Tyto subrutiny představují elementární bloky operačního systému pro přepínání kontextu, přerušení, výjimky a správu paměti. Jednotlivé subrutiny je možno volat hardwarově nebo instrukcemi CALL_PAL. PALcode je napsán ve standardním assembleru Alpha AXP s některými implementačně závislými rozšířeními tak, aby

umožňoval přímý přístup k nízkoúrovňovým hardwarovým funkcím, například interním registrům procesoru. PALcode se provádí v režimu PALmode, privilegovaném režimu, který zabrání vzniku některých systémových událostí a umožňuje PALcodu převzít řízení nad hardwarem fyzického systému.

Dodatek

C

Užitečné adresy WWW a FTP

Dále uvádíme seznam užitečných adres WWW a FTP:

http://www.azstarnet.com/~axplinux

Jedná se o stránky Davida Mosbergera-Tanga věnované Alpha AXP Linuxu, na tomto místě naleznete odkazy na všechny dokumenty HOWTO týkající se procesoru Alpha AXP. Obsahuje také řadu odkazů na další informace o Linuxu a procesorech Alpha AXP, například datové tabulky procesoru.

http://www.redhat.com/

Webové stránky Red Hat. Obsahují řadu užitečných odkazů.

ftp://sunsite.unc.edu

Významná síť se spoustou zdarma dostupných programů. Programy pro Linux naleznete v adresáři *pub/Linux*.

http://www.intel.com

Domovská stránka firmy Intel, dobré místo k nalezení informací o čipech Intel.

http://www.ssc.com/lj/index.html

Linux Journal je velmi dobrý časopis věnovaný Linuxu a díky řadě skvělých článků se předplatné rozhodně vyplatí.

http://www.blackdown.org/java-linux.html

Primární síť informací o Javě na Linuxu.

ftp://tsx-11.mit.edu/~ftp/pub/linux

FTP server Linuxu na MITu.

ftp://ftp.cs.helsinki.fi/pub/Software/Linux/Kernel

Zdrojové kódy jádra Linuxu.

http://www.linux.org.uk

Linux User Group ve Velké Británii.

http://sunsite.unc.edu/mdw/linux.html

Domovská stránka Linux Documentation Projectu.

http://www.digital.com

Domovská stránka firmy Digital Equipment Corporation.

http://altavista.digital.com

Vyhledávací stroj Altavista firmy Digital. Velmi dobré místo pro hledání informací na webu.

http://www.linuxhq.com

Stránky Linux HQ obsahují aktuální oficiální i neoficiální patche a dále rady a odkazy, které vám mohou pomoci získat ty nejvhodnější zdrojové kódy jádra pro váš systém.

http://www.amd.com

Domovské stránky firmy AMD.

http://www.cyrix.com

Domovské stránky firmy Cyrix.

Dodatek

D

Slovníček

ARP — Address Resolution Protocol. Slouží k překladu IP adres na fyzické hardwarové adresy.

ASCII — American Standard Code for Information Interchange. Každé písmeno abecedy je reprezentováno 8bitovou hodnotou. ASCII kód se často používá k ukládání psaných textů.

Bajt — 8 bitů dat.

Bit — Elementární jednotka dat, která je reprezentována jako 0 nebo jako 1 (zapnuto nebo vypnuto).

Bottom-half handler — Obslužný systém pro práci s úlohami řazenými ve frontách.

C — Programovací jazyk vysoké úrovně. Většina jádra Linuxu je napsána v jazyce C.

CPU — Central Processing Unit. Hlavní zařízení počítače, viz též *procesor* a *mikroprocesor*.

Datová struktura — Skupina dat v paměti uspořádaná do položek.

DMA — Direct Memory Access. Přímý přístup do paměti.

EIDE — Extended IDE.

ELF — Executable and Linkable Format. Objektový formát souborů navržený v Unix System Laboratories je dnes uznáván jako nejčastěji používaný formát v Linuxu.

Fronta úloh Mechanismus pro odkládání úloh v Linuxu.

Funkce Kus programu zajišťující nějakou činnost, například vrácení většího ze dvou čísel.

IDE Integrated Disk Electronics.

IP Internet Protocol.

IPC Interprocess Communication, meziprocesová komunikace.

IRQ Interrupt Request Queue, fronta požadavků na přerušení.

ISA Industry Standard Architecture. Dnes poněkud zastaralý standard rozhraní datové sběrnice pro komponenty jako řadič disketových mechanik.

Kilobajt Tisíc bajtů dat (přesně 1 024), často se zapisuje jako KB.

Megabajt Milión bajtů dat (přesně 1 048 576 B nebo 1 024 KB), zapisuje se jako MB.

Mikroprocesor Integrovaný CPU. Většina moderních CPU jsou *mikroprocesory*.

Modul jádra Dynamicky nahrávaná funkce jádra jako například souborový systém nebo ovladač zařízení.

Modul Soubor obsahující instrukce CPU buď v podobě instrukcí assembleru, nebo instrukcí vysokoúrovňového jazyka jako je například C.

Objektový soubor Soubor obsahující strojový kód a data, který ještě nebyl slinkován s dalšími objektovými soubory a knihovnami do *spustitelného obrazu*.

Obraz Viz *Spustitelný obraz*.

Ovladač zařízení Program řídící určité zařízení, například ovladač zařízení NCR 810 slouží k řízení řadiče SCSI NCR 810.

Parametr Funkce a rutiny přebírají parametry, které zpracovávají.

PCI Peripheral Component Interconnect. Standard popisující jak se vzájemně propojují komponenty počítačového systému.

Periferie Inteligentní procesor pracující místo procesoru systému. Například čip řadiče IDE.

Proces Entita, která může provádět *program*. Proces si je možno představit jako *program* v akci.

Procesor Zkrácený název pro mikroprocesor, ekvivalentní termín pro *CPU*.

Program Koherentní množina instrukcí procesoru, která provádí nějakou úlohu, například vytiskne text „hello world". Viz též *spustitelný obraz*.

Protokol Síťový *jazyk* používaný k přenosu aplikačních dat mezi dvěma spolupracujícími procesy nebo síťovými vrstvami.

Registr Místo na čipu, používané k uložení informací nebo instrukcí.

Rozhraní Standardní metoda volání rutin a předávání datových struktur. Například rozhraní mezi dvěma vrstvami kódu může být popsáno rutinami, které přebírají a vracejí určité datové struktury. Dobrým příkladem rozhraní je virtuální souborový systém Linuxu.

Rutina Podobá se funkci s tou výjimkou, že, přísně vzato, nevrací výslednou hodnotu.

SCSI Small Computer Systems Interface.

Shell Program působící jako rozhraní mezi operačním systémem a živým uživatelem. Označuje se také jako *příkazový interpret*, v Linuxu se nejčastěji používá shell *bash*.

SMP Symetrický multiprocessing. Systém, v němž je více než jeden procesor, a ty si mezi sebou spravedlivě rozdělují práci.

Software CPU instrukce (ať už v assembleru, nebo ve vyšším jazyce) a data. Prakticky ekvivalent pojmu *program*.

Soket Reprezentuje jeden konec síťového spojení. Linux podporuje soketové rozhraní BSD.

Spustitelný obraz Strukturovaný soubor obsahující strojové instrukce a data. Soubor je možno nahrát do virtuální paměti procesu a spustit. Viz též *program*.

Stránka Fyzická paměť se dělí na stejně velké stránky.

System V Klon Unixu uvolněný v roce 1983, který, mimo jiné, obsahoval mechanismy meziprocesové komunikace.

TCP Transmission Control Protocol.

UDP User Datagram Protocol.

Ukazatel Místo v paměti, které obsahuje adresu jiného místa v paměti.

Virtuální paměť Hardwarové a softwarové mechanismy zajišťující, že fyzická paměť v systému se jeví větší, než jaká doopravdy je.

Příloha

Seznam literatury

Richard L. Sites: *Alpha Architecture Reference Manual*, Digital Press

Matt Welsh, Lar Kaufman: *Running Linux*, O'Reilly&Associates

PCI Special Interest Group: *PCI Local Bus Specification*

PCI Special Interest Group: *PCI BIOS ROM Specification*

PCI Special Interest Group: *PCI to PCI Bridge Architecture Specification*

Intel: *Peripheral Components*, Intel 296467

Brian W. Kernihan, Dennis M. Ritchie: *The C Programming Language*, Prentice Hall

Steven Levy: *Hackers*, Penguin Books

Intel: *Intel486 Processor Family: Programmer's Reference Manual*, Intel

Comer D. E.: *Internetworking with TCP/IP, Volume 1 - Principles, Protocols and Architecture*, Prentice Hall

David Jagger: *ARM Architectural Reference Manual*, Prentice Hall