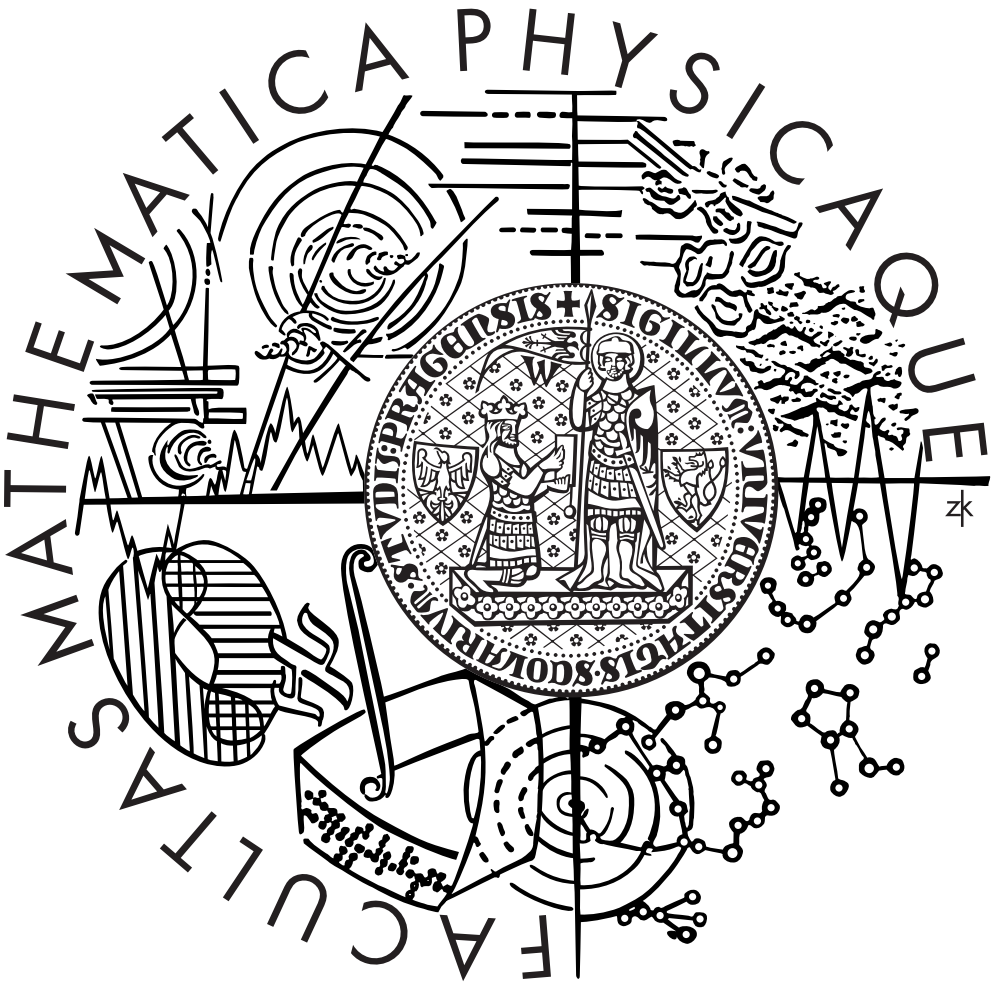
Universita Karlova v Praze
Matematicko-fyzikální fakulta

Diplomová práce



Mikuláš Patočka

Srovnávací studie jádra Linuxu a FreeBSD

Katedra softwarového inženýrství

Vedoucí diplomové práce: Mgr. Jakub Yaghob

Studijní program: Informatika, softwarové systémy, počítačové systémy

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 7.1.2003 Mikuláš Patočka

Obsah

Název práce: Srovnávací studie jádra Linuxu a FreeBSD

Autor: Mikuláš Patočka

Katedra (ústav): Katedra softwarového inženýrství

Vedoucí diplomové práce: Mgr. Jakub Yaghob

e-mail vedoucího: yaghob@ulita.ms.mff.cuni.cz

Abstrakt: Cílem práce je popsat strukturu a funkčnost jader operačních systémů Linux a FreeBSD. Práce se zaměřuje na popis synchronizačních mechanismů jader, podporu víceprocesorových systémů (SMP), mechanismus alokace a mapování paměti v jádře, scheduler, rozhraní mezi jádrem a ovladači filesystému (VFS), filesystémy, strukturu síťových bufferů a rozšíření unixového rozhraní mezi jádrem a procesy. Neexistuje obecné rozhodnutí, zda je lepší Linux, nebo FreeBSD — pro některé typy úloh se více hodí Linux, pro jiné FreeBSD.

V práci je ukázáno, že FreeBSD má lepší maskování přerušení než Linux. Na Linuxu je lepší podpora víceprocesorových systémů. FreeBSD má omezení 4G RAM, nicméně s touto pamětí nakládá lépe než Linux 2.4. Linux 2.5 má lepší scheduler než FreeBSD, které má lepší scheduler než předchozí verze Linuxu. Linux podporuje více filesystémů, některé mají žurnálování; FreeBSD používá soft-updates k udržování konzistence filesystému. Linux používá POSIX realtimové signály pro čekání na více událostí, FreeBSD má vlastní mechanismus kqueue, který je lepší. Linux 2.5 má nové rozhraní epoll, které odstraňuje nevýhody realtimových signálů.

Klíčová slova: Linux, FreeBSD, operační systém, jádro

Title: Comparison of Linux and FreeBSD kernels

Author: Mikuláš Patočka

Department: Department of software engeneering

Supervisor: Mgr. Jakub Yaghob

Supervisor's e-mail address: yaghob@ulita.ms.mff.cuni.cz

Abstract: The purpose of this work is to describe the structure and functionality of Linux and FreeBSD kernels. The work focuses on description of synchronisation primitives, the support for multiprocessor systems, allocation and mapping of kernel memory, scheduler, the interface between kernel and filesystem drivers (VFS), filesystems, the structure of networking buffers and extensions to Unix application program interface. There is no general decision whether Linux or FreeBSD is better — Linux is more suitable for some tasks and FreeBSD for other.

The work shows that FreeBSD has better interrupt masking than Linux. Linux has better support for multiprocessor systems. FreeBSD has limit of 4G RAM, but it uses this memory more efficiently than Linux 2.4. Linux 2.5 has better scheduler than FreeBSD which has better scheduler than previous versions of Linux. Linux supports more filesystems, some are journalled; FreeBSD uses soft-updates to maintain filesystem consistency. Linux uses POSIX realtime signals to wait for multiple events, FreeBSD has its own interface kqueue which is better. Linux 2.5 has new interface epoll that removes some shortcomings of realtime signals.

Keywords: Linux, FreeBSD, operating system, kernel

1. Úvod

V posledních několika letech došlo k velkému rozvoji výpočetní techniky. Před pěti lety málokdo předpokládal, že dnes budou běžně používané počítače s více gigahertzovými procesory, několika gigabyty paměti a připojené na gigabitový ethernet. Bylo nutno těmto zvýšeným nárokům na hardware přizpůsobit operační systémy a optimalizovat je, aby se pod vysokou zátěží na rychlém hardwaru efektivně chovaly. Cílem této práce je porovnání jader současných verzí operačních systémů Linux a FreeBSD a ukázání, jak se jejich vývojářům podařilo se se zvýšenými nároky vyrovnat. Tyto systémy jsem vybral, protože jsou ve své kategorii nejlepší a mají dostupný zdrojový kód pod free[1]licencemi. Komerčními systémy se zabývat nebudu, protože jejich zdrojový kód není dostupný nebo je dostupný pouze pod velmi omezujícími licenčními podmínkami. V určitých částech práce se zmiňuji i o tom, jak je ten který problém řešen na některém komerčním systému, nicméně tyto informace je třeba brát bez záruky — nikdo, kromě zaměstnanců firmy vyvíjející daný systém, nemůže vědět, zda je to tak v systému skutečně implementováno, nebo ne.

Cílem této práce je popsat struktury jader současných systémů a algoritmy v nich použité a ukázat, jakým způsobem se vývojáři vypořádali s problémy vyskytujícími se v oblasti návrhu operačních systémů. Tuto práci jsem se rozhodl napsat proto, že žádná podobná práce, která by popisovala vnitřnosti operačních systémů, neexistuje. Literatura v oblasti operačních systémů je omezena na uživatelské nebo programátorské návody (výjimečně se dá někdy nalézt návod, jak napsat ovladač zařízení do jádra, ale to je asi tak všechno) a sami programátoři o své práci články nepublikují.

Cílem této práce není dělat benchmarky — benchmark vždy testuje jen jednu vlastnost, zatímco při běžném provozu jsou vytíženy všechny části systému. Pokud například v nějakém benchmarku vyjde, že systém X umí otevřít a přečíst soubor třikrát rychleji než systém Y, v praxi to má zanedbatelný efekt, neboť systém dělá spoustu jiných operací, než jen otevírání a čtení souborů. Dalším problémem benchmarků je ne příliš deterministické chování procesorových cachí. „Úzkým hrdlem“, které nejvíce zpomaluje rychlost počítače, není malá frekvence procesoru, nýbrž sběrnice mezi procesorem, L2 cachí a hlavní pamětí[2]. V praxi se pak počítač chová tak, že pokud trochu změníme pořadí funkcí nebo proměnných v paměti, data se hůře nebo lépe vejdou do cache[3]a rych-

[1] Pojem „free“ neznamená, že software je zadarmo — pojmem „Free software“ je myšleno, že každý člověk má právo program nejen používat, ale i zkoumat zdrojové kódy, měnit je a mít možnost distribuovat modifikované verze. O filosofii free software je možno se dovědět více na stránkách Free Software Foundation `www.fsf.org`. Nachází se tam rovněž přesná definice, co ještě je a co už není free software. Někdy se místo pojmu „Free software“ používá termín „Open source software“, který má podobný význam.

[2] Čtení z L1 cache na procesoru trvá jeden takt. Čtení z L2 cache trvá několik taktů. Čtení z hlavní paměti trvá několik desítek taktů. Je vidět, že rychlost sběrnice je mnohdy důležitější než samotný výkon procesoru.

[3] Struktura cache a optimalizace programování pro lepší využití cache bude v této práci později popsána.

lost programu se může třeba dvakrát zvětšit nebo zmenšit. Výsledek benchmarku +/– dvakrát v podstatě neznamená vůbec nic — v jiné konfiguraci to může vyjít naopak.

V práci bylo několik benchmarků uděláno, byly dělány na počítači Pentium-MMX 200MHz, 160M RAM. Testovány byly systémy Linux 2.4.20 a FreeBSD 4.7. K dělání benchmarků se nehodí experimentální jádra systémů, neboť ta obsahují spoustu debuggovacího kódu, který zpomaluje a výsledky benchmarků zkresluje.

1.1. Jiné volně šiřitelné systémy

Existuje více operačních systémů pod free licencemi. Původní BSD se rozdělilo na tři nyní paralelně vyvíjené verze: NetBSD, OpenBSD a FreeBSD. NetBSD se zaměřeno na maximální portabilitu — je to operační systém, který funguje na největším množství architektur. Vývojáři OpenBSD se snaží o maximální bezpečnost. OpenBSD je vyvíjeno mimo USA, takže jeho zabezpečovací funkce nebyly ohrožovány donedávna platnými americkými zákony proti šifrování. Vývojáři FreeBSD odstranili portabilitu a snaží se maximalizovat výkon na procesorech Intel (nedávno přibyla Alpha a v nové verzi se chystá port i na IA64 a možná i několik dalších architektur).

K rozdělení Linuxu zatím nedošlo — jádro stále drží pod kontrolou zakladatel projektu Linus Torvalds. Existují různé modifikované verze Linuxu, např. Beowulf, což je systém pro clusterování, nebo rtLinux, což je jádro Linuxu modifikované, aby běželo nad realtimovým mikrokernelem, schopné obsluhovat zařízení vyžadující přesnou dobu odezvy (děje se tak samozřejmě na úkor rychlosti běžné práce systému — mikrokernely jsou vždy pomalejší než monolitická jádra). Na Linux taktéž existuje veliké množství modifikací (tzv. patchů), které systém rozšiřují o některé nové funkce. Patche systém rozšiřují o nové schopnosti, na druhou stranu však také mohou obsahovat chyby vedoucí k pádu systému.

Kromě Linuxu a BSD existují mezi free softwarem i další menší operační systémy. Asi nejznámější a nejdále dotáhnutý je AtheOS vyvinutý Kurtem Skauenem. Je to monolitické jádro unixového typu, nad kterým běží zcela nové objektově orientované multithreadové grafické rozhraní. Poté, co Kurt nebyl příliš aktivní při vývoji a údržbě jádra, jiná skupina vývojářů přejmenovala AtheOS na Syllable a pokračuje v jeho vývoji.

Existují i volně šiřitelné mikrokernelové systémy — GNU Hurd je systém serverů postavený nad mikrojádrem Mach. Hurd jsem kdysi zkoušel a je velmi pomalý — kopírování 12M souboru trvalo 45 sekund (Pentium 200, 32M ram). Jiným volně šiřitelným mikrokernelovým systémem je VSTA. Mikrokernelové systémy se v dnešní době již nevyvíjejí, neboť jsou pomalé a komunikace mezi jednotlivými procesy zajišťujícími služby systému je příliš komplikovaná.

2. Historie

2.1. Historie FreeBSD

FreeBSD pochází ještě z klasického Unixu. První verzi Unixu vyvinuli roku 1969 na (v té době zastaralém) PDP-7 v Bellových laboratořích Ken Thompson, Dennis Ritchie a Brian Kerninghan. Filesystém Unixu byl podobný dnešnímu filesystému — měl pevné místo pro inody, hierarchickou adresářovou strukturu, adresáře byly uloženy stejným způsobem jako soubory. Unix uměl obsluhovat dva terminály a ke každému terminálu existoval jeden proces — v systému byl tedy fixní počet dvou procesů a `fork` neexistoval. Syscall `exit` pouze smazal aktuální paměť procesu a natáhl na její místo shell. PDP-7 mělo lineární paměť — nemělo segmentaci, stránkování ani jiný prostředek ochrany paměti — multitasking mezi procesy dělal Unix swapováním celých procesů na disk. Po čase do Unixu přibyly syscally `fork`, `exec` a dnešní podoba `exit`. Procesy však stále zůstávaly bez ochrany paměti.

K vývoji Unixu se podařilo získat nové PDP-11, Unix byl přepsán pro tuto architekturu, přibyly pipy, začalo přepisování do C a byla implementována ochrana paměti pomocí segmentace (stále se provádělo swapování celých procesů — nyní však již v paměti mohlo být paralelně zavedeno více procesů). Verze 5 Unixu byla již kompletně napsaná v jazyce C. Ve verzi 7 byl příkaz `chdir` změněn na `cd` a systém se ovládal v podstatě úplně stejně jako dnešní unixy.

Bellovy laboratoře byla telefonní společnost, nikoli softwarová firma, proto manažeři dovolili uvolnit Unix na university pod ne příliš omezujícími podmínkami. University mohly získat zdrojový kód, na něj vyrábět patche a tyto patche šířit. Jednou z universit, kam se Unix dostal, byla universita v Berkeley v Kalifornii. Zde začali vyvíjet svoji sadu patchů označovanou jako BSD — Berkeley System Distribution. Původní BSD nebyl operační systém — byla to sada patchů, které do klasického Unixu přidávaly některé nové možnosti, jako například stránkování a virtuální paměť, komunikaci pomocí TCP/IP socketů, možnost pracovat s více odlišnými typy filesystému a mnohé další. V release BSD 4.4 se podařilo v podstatě už téměř celý původní Unix přepsat a systém BSD se (jako produkt university) stal volně šiřitelný.

Bill Jolitz se svou manželkou Lynne začali tento systém portovat na architekturu Intel 386 a tak vzniklo 386BSD. Poté, co Bill začal vývoj zanedbávat, začaly se objevovat patche, a když už patchů bylo příliš, Jordan Hubbard, Nate Williams a Rod Grimes založili FreeBSD, což bylo v podstatě 386BSD se spoustou patchů, které se za tu dobu naskládaly. Vývoj FreeBSD dále pokračoval.

Firma Novell, které mezitím připadla licence na původní Unix, zažalovala universitu v Berkeley, že přepsáním Unixu a uvolněním BSD porušila autorská práva. Universita hned nato vznesla žalobu proti Novellu za to, že rozšíření vyvinutá v Berkeley, jako například TCP/IP, používá ve svém Unixu, aniž by dodržel podmínky licence, pod kterou bylo BSD vydáno. Obě instituce se nakonec usmířily, stáhly žaloby a bylo vydáno nové BSD 4.4-Lite, ze kterého byly odstraněny některé sporné části a které bylo uznáno za právně čisté.

FreeBSD 2.0 bylo přepsáno, aby bylo založeno na čistém „lite“ release a aby neobsa-

hovalo žádný kód, který se nacházel v BSD 4.4, ale byl odstraněn v BSD 4.4-Lite.

Vývoj FreeBSD pak dále pokračoval — docházelo ke změnám ve virtuální paměti, filesystému, síťování a ke zvyšování výkonu. FreeBSD je vyvíjeno pomocí systému CVS. Existují „větve“ (anglicky branches) — pro každé hlavní číslo jedna — nyní například existují větve 2.2-STABLE, 3-STABLE, 4-STABLE, 5-CURRENT. Větve 2.2 a 3 se již nevyvíjejí. Větve je možno stáhnout pomocí CVS. Z každé vyvíjené větve se čas od času udělá RELEASE — v současné době jsou aktuální releasy 4.6-STABLE nebo 5.0-DEVELOPER PREVIEW. Větev 4 je stabilní a je vhodná pro produktivní použití. Do této větve se nedávají žádné nové vlastnosti; pouze se v ní opravují chyby. Větev 5 je experimentální, obsahuje nové ne příliš dobře otestované vlastnosti a doporučuje se používat pouze vývojářům. Až budou chyby ve větvi 5 odstraněny, stane se tato novou stabilní větví a větev 4 se přestane vyvíjet.

2.2. Historie Linuxu

Linux začal jako osobní projekt studenta Linuse Torvaldse. Linusovi se nelíbil mikrokernelový systém Minix, vyvinutý profesorem Andrewem Tanenbaumem, tehdy používaný k výuce operačních systémů. Minix běžel na 8086 a 80286, jednotlivé komponenty jádra byly samostatné procesy a ke komunikaci používaly fronty zpráv. V praxi to nemělo moc dobrý výkon, neboť pokud například proces obsluhující filesystém čekal na data z diskety, nemohl současně odpovídat na jiné požadavky.

První verze Linuxu 0.01 byla vydána v roce 1991. Linux byl kompletně navržen pro 80386, návrh jádra byl (a dodnes je) podobný návrhu klasického Unixu (monolitické jádro, syscally jsou prostá volání do jádra, žádné mikrokernelové „servery“ ani fronty zpráv). Linux měl filesystém kompatibilní s Minixem (a dodnes jádra Linuxu s minixovým filesystémem umějí pracovat), bufferovou cache, ovladač pouze pro AT harddisk. Linux 0.01 neuměl swapovat ani mapovat soubory do paměti, ale měl sdílení stránek po provedení `fork` a copy-on-write. Linux uměl komunikovat přes sériové porty a měl napevno „zadrátovanou“ finskou klávesnici.

Poté byl Linux postupně rozšiřován — přišly nové filesystémy (Xia, Ext, Ext2), které odstranily omezení minixového filesystému na velikost partition a délku názvu souboru. Byla napsána komunikace pomocí TCP/IP, správa paměti byla rozšířena o swapování, sdílení stránek programů a mapování souborů do paměti. Ve verzi 1.0 byl Linux již plnohodnotný systém.

Od verze 1.0 začalo specifické číslování verzí Linuxu. Pokud je druhé číslo verze sudé, jedná se o stabilní jádro vhodné pro použití v produktivních systémech. Ve stabilních jádrech se pouze opravují chyby, výjimečně se tam přidávají nové ovladače, nikdy se nedělají zásadní změny struktury subsystémů. Pokud je druhé číslo verze liché, jedná se o experimentální verzi. V experimentálních verzích se testují nové vlastnosti, experimentální jádra mohou obsahovat chyby (někdy mohou způsobit i ztrátu dat) a jsou vhodná pro vývojáře. Až se chyby v experimentálním jádře odstraní, přechází toto do jádra stabilního: například 1.1 přešlo do 1.2, 1.3 přešlo do 2.0 apod.

Ve verzi 2.0 přišly moduly jádra — ovladače bylo možno natahovat a odstraňovat za běhu bez nutnosti překompilování jádra a rebootu počítače. Verze 2.0 rovněž měla

page-cache, která sjednotila a zefektivnila správu paměti. 2.0 mohlo fungovat na více-procesorových strojích, ale do jádra se vždy mohl dostat jen jeden procesor — což bylo neefektivní, pokud velké množství procesů volalo služby jádra. V 2.2 bylo toto omezení částečně odstraněno.

3. Struktura jader systémů

Linux i FreeBSD používají dvě úrovně privilegovanosti procesoru. Tyto úrovně nazvěme USER a KERNEL mód[1]. V USER módu procesor nedovolí vykonávání privilegovaných instrukcí, které by mohly způsobit pád systému, taktéž je v něm omezen přístup do paměti a zpravidla je úplně zakázána komunikace na IO portech. V KERNEL módu naproti tomu přístup není nijak omezen. Z KERNEL módu do USER módu se procesor přepne pomocí speciální instrukce (tato instrukce je privilegovaná — nejde ji tedy provádět v USER módu). Z USER módu do KERNEL módu se procesor přepne buď přes výjimku (vykonání nedovolené instrukce, přístup na nedovolenou adresu), nebo při příchodu přerušení od nějakého zařízení. Při přístupu k paměti používají procesory mechanismus stránkování. Když se v programu vyskytne přístup na nějakou adresu (nazývá se virtuální adresa), je k této adrese v tabulce stránek nalezena odpovídající fyzická adresa, a na tuto fyzickou adresu je pak přistoupeno do paměti. Pokud mapování virtuální na fyzickou adresu není v tabulce nalezeno, vyvolá se výjimka. V tabulce stránek se taktéž nalézají práva přístupu — čili úroveň privilegovanosti, od které je možno na danou adresu přistoupit, a druh přístupu (čtení, zápis, někde i pouštění kódu). Aby se nemusela při každém přístupu do paměti procházet tabulka, má procesor cache na několik posledních mapování. Tato cache se nazývá TLB (translation lookaside buffer). Tabulka stránek má většinou strukturu stromu tabulek o pevné výšce (například procesory IA32 mají dvouúrovňové tabulky, Pentium PRO, Pentium 2 a vyšší je možno přepnout do speciálního režimu, kde jsou tabulky trojúrovňové). Tabulka nejvyšší úrovně se nazývá adresář stránek a ukazuje na ni speciální registr procesoru (instrukce přístupu k tomuto registru jsou privilegované, takže kód běžící v USER módu nemůže s mapováním paměti manipulovat). Některé architektury, jako například Sparc64, nemají vůbec tabulku stránek a mají pouze TLB, která je plněna pomocí speciálních instrukcí.

Linux i FreeBSD byly navrženy tak, že všechny procesy běží v USER módu a celé jádro běží v KERNEL módu. Každý proces má svoji vlastní tabulku stránek, která určuje jeho adresový prostor. Jádro se nachází na virtuálních adresách pro ně rezervovaných (a přístupných pouze z KERNEL módu). Kód i data celého jádra jsou sdíleny mezi všemi procesy a jsou z KERNEL módu vždy přístupné. Taková struktura jádra se nazývá monolitické jádro. Výhoda monolitického jádra je jednoduchost (ke všem strukturám je vždy přístup) a rychlost (při práci jádra není nutno přepínat tabulky stránek, což je velmi pomalé). Nevýhoda je naproti tomu taková, že když se v jakékoli části jádra vyskytne chyba, znamená to pád celého systému. Na chyby v jádře systémy reagují různě: Linux při chybě jádra vypíše informaci o stavu systému a registrech procesoru na konzoli (oblíbená hláška „`Unable to handle kernel paging request. Oops!`“). Jádro Linuxu následně ukončí aktuální proces a snaží se pokračovat. FreeBSD při chybě jádra vypíše hlášku na konzoli, zastaví celý systém a případně (pokud to bylo nastaveno) provede ještě dump celé paměti na diskovou partition k tomuto účelu připravenou. Každý z těchto dvou způsobů obsluhy chyb má své výhody i nevýhody; nelze obecně říct, který

[1] Existují i systémy, které používají více úrovní privilegovanosti — například VMS má čtyři: KERNEL, EXECUTIVE, SUPERVISOR a USER. Multics měl dokonce osm úrovní.

je lepší. Když dojde k nějaké drobné chybě, Linux se z toho dokáže jakž takž zotavit, naproti tomu FreeBSD se celé zastaví. Na druhou stranu, pokud je chyba závažnější, nepodaří se je většinou v Linuxu odstranit ukončením procesu, opakuje se znovu a znovu v různých procesech a z běžících hlášek na konzoli není možno určit tu první ani její příčinu. FreeBSD se v tom případě chová lépe, neboť se zastaví hned na první chybě a případně ještě provede dump paměti, z čehož je možno lépe zjistit příčinu chyby a chybu odstranit. Pro zpracování závažných chyb oba systémy obsahují funkci `panic`, která vypíše chybovou hlášku na terminál a zastaví celý systém. Funkce se volá, pokud nějaká část jádra zjistí, že došlo k poškození jejích dat a že již není možno pokračovat.

Linux ani FreeBSD nepoužívají segmentaci, pracují v lineárním adresním prostoru. Segmentace (tak, jak je například implementovaná na procesorech IA32) je velmi těžkopádně použitelná — použití segmentace by znamenalo zvětšení pointerů ze 4 na 6 bytů a s tím související veliké množství práce s pointery a se segmentovými registry. V podstatě žádný 32bitový operační systém na IA32 segmentaci nepoužívá.

4. Synchronizační mechanismy v jádrech

4.1. Přepínání procesů

Nyní se zaměřím na funkci jader na jednoprocesorových systémech — rozšíření pro víceprocesorové systémy bude popsáno v následující kapitole.

Každý proces může běžet v USER módu, při obsluze syscallu nebo zpracování výjimky se dostane do KERNEL módu. V USER módu existuje preemptivní multitasking — t.j. k přepnutí procesu dojde kdykoli, kdy to scheduler uzná za vhodné. Naproti tomu při běhu procesu v KERNEL módu se používá kooperativní multitasking — t.j. proces je možno přepnout pouze tehdy, když o to sám požádá nebo když čeká na nějakou událost (například data z disku, ze sítě, stisk klávesy a podobně). Takové čekání na nějakou událost se nazývá *zablokování procesu*. Při zablokování dojde k přepnutí procesu a na procesoru může běžet jiný proces, který není zablokovaný. Existují speciální procesy zvané *kernel thready*, které nemají vlastní tabulku stránek ani uživatelský adresový prostor a běží po celou dobu v jádře. Kernel thready se používají k vykonávání nějakých činností na pozadí, které nesouvisejí s žádným uživatelským procesem (například zapisování modifikovaných bufferů na disk nebo swapování). V Linuxu se možné přepnutí procesu v jádře dělá funkcí `void cond_resched()` nebo `schedule()`, na FreeBSD funkcí `void mi_switch()`. Výhodou kooperativního multitaskingu je, že zjednodušuje návrh jádra, nevýhodou jsou občasné pomalejší reakce systému. Při běžném provozu se to téměř nepozná, ale realtimové aplikace vyžadující garantované probuzení a spuštění procesu do určitého času na Linuxu ani FreeBSD běžet nemohou. V Linuxu byla snaha tento problém řešit a byly použity dvě metody:

- Byla snaha nalézt v jádře všechny déletrvající cykly a do nich bylo vloženo podmíněné přepnutí procesu pomocí `cond_resched`. Toto řešení se nazývá low-latency patch.
- Druhou metodou je preemptivní kernel — čili možnost libovolného přepínání procesů i při běhu v jádře. Značně to souvisí s implementací podpory pro víceprocesorové systémy, a proto způsob implementace popíšu až v následující kapitole.

  Obě tato řešení se nacházejí pouze v experimentální verzi Linuxu 2.5.

Závěr: Ani jedno z jader neposkytuje garantované probuzení procesu v určitém čase. V Linuxu je snaha tento problém řešit, ale řešení se nacházejí zatím pouze v experimentálních verzích.

4.2. Přerušení

Různá zařízení generují přerušení. Přerušení se obsluhují při běhu v USER i KERNEL módu. Po skončení obsluhy přerušení se řízení vrací na místo, kde k přerušení nastalo. Často se stává, že přerušení způsobí probuzení nějakého procesu — pokud takové přerušení nastalo v USER módu, nový proces se spustí okamžitě po skončení obsluhy přerušení. Pokud nastalo v KERNEL módu, proces se přepne při nejbližším přechodu do USER módu nebo při zavolání výše popsané funkce pro kooperativní schedulování.

Obsluha přerušení s sebou přináší problém: přerušení mohou chodit současně, nebo

může jedno přerušení přijít v průběhu jiného. Některá přerušení mohou trvat déle. Existují zařízení, která potřebují přerušení obsloužit rychle, jinak dochází k chybám nebo degradaci výkonu (jedná se zejména o méně kvalitní zařízení: zvuková karta s malými buffery, nedostane-li včas data, začne vydávat praskavé zvuky; síťová karta, jejíž přerušení není dostatečně rychle obslouženo, začne ztrácet packety; bude-li přerušení od disku příliš dlouho čekat, poklesne přenosová rychlost disku). Kvalitnější zařízení na latenci přerušení moc závislá nejsou: kvalitní zvukové karty mají velikou vyrovnávací paměť, kvalitní síťové karty mají velikou paměť pro odchozí i příchozí packety, kvalitní SCSI disky mají tagovanou frontu a jsou schopny přijmout a vykonávat několik požadavků současně bez jakékoli účasti procesoru.

Pro jakýkoli kód, který je volán z obsluhy přerušení, platí omezení: kód nesmí provést zablokování (ani volat jakoukoli funkci, která by zablokování mohla provést). Vzhledem k tomu, že obslužná rutina přerušení běží na zásobníku nějakého procesu, který zrovna běžel, když se přerušení vyskytlo, a který s tímto přerušením vůbec nesouvisí, nemělo by blokování tohoto procesu žádný smysl. Nyní popíšu způsoby, jakými oba systémy přerušení zpracovávají:

Linux zpracovává přerušení následovně: když se vyskytne přerušení, je na řadiči zamaskováno, pak se povolí všechna přerušení na procesoru a vykoná se obslužná rutina přerušení. Obslužná rutina tedy může být přerušena jinými přerušeními. Ovladač může při registraci přerušení vynutit, aby při volání obslužné rutiny tohoto přerušení byla přerušení na procesoru zamaskovaná. Registrace přerušení se provádí funkcí `int request_irq(unsigned int irq, void (*handler)(int, void *, struct pt_regs *), unsigned long irqflags, const char *devname, void *dev_id)`. První parametr je číslo přerušení, druhý je pointer na obslužnou rutinu, třetí jsou příznaky `SA_SHIRQ` (pokud přerušení lze sdílet mezi několika zařízeními), `SA_INTERRUPT` (pokud chceme zakázat ostatní přerušení po dobu vykonávání obslužné rutiny), `SA_SAMPLE_RANDOM` (pokud je přerušení použitelné pro generátor náhodných čísel)[1], čtvrtý parametr je název zařízení, který se objeví ve výpisu přerušení, a pátý parametr je pointer, který je předáván obslužné rutině.

Pokud ovladač zařízení pracuje se strukturami, které mohou být přerušením modifikované, musí toto přerušení zakázat. Linux vynucuje v takovém případě zakázání všech přerušení (zakázání jednoho přerušení na řadiči je moc pomalé a softwarové maskování daného přerušení Linux neumí). Přerušení je možno zakázat pomocí maker `cli()` a `sti()`. Makro `save_flags(x)` uloží aktuální stav zakázání/povolení do proměnné `x`, která musí být typu `unsigned long`. Makro `restore_flags(x)` tento stav zase obnoví. Poslední dvě makra se používají ve funkcích, které potřebují zakázat přerušení, ale není jasné, jestli už přerušení nejsou zakázána. Podobně existují makra `__cli()`, `__sti()`, `__save_flags(x)` a `__restore_flags(x)`, která provádějí zakazování a povolování přerušení pouze na tom

[1] Tento příznak je třeba nastavit u náhodně přicházejících přerušení (například z klávesnice nebo myši) a nenastavovat ho u periodicky přicházejících přerušení (například od zvukové karty). Kvalitní generátor náhodných čísel je potřeba pro správné generování sekvenčních čísel v TCP spojení — kdyby čísla byla předvídatelná, ohrozilo by to bezpečnost TCP/IP.

procesoru, na kterém kód právě běží. Pokud jsou přerušení zamaskována, proces se v Linuxu nesmí zablokovat.

Časté zakazování všech přerušení a nemožnost selektivního zakázání několika přerušení je problém Linuxu. Na Linuxu se přerušení zakazují v podstatě kdykoli je potřeba provádět nějakou operaci se strukturami, které mohou být nějakým přerušením modifikovány. Bohužel se zakazují například i při psaní na konzoli (neboť kód z přerušení může na konzoli zapisovat), posílání požadavků na IDE disk a spoustě jiných příležitostí. To vede k veliké latenci přerušení a důsledek je například takový, že se ztrácejí packety na paralelní nebo sériové lince, například pokud uživatel přepne konzoli.

Ve FreeBSD je problém přerušení řešen lépe. FreeBSD téměř nikdy nezakazuje přerušení na procesoru, ale používá softwarové maskování jednotlivých přerušení. Přerušení je registrováno pomocí funkce `int bus_setup_intr(device_t dev, struct resource *r, int flags, driver_intr_t handler, void *arg, void **cookiep)` (předtím muselo být ještě alokováno pomocí `bus_alloc_resource`. První parametr je pointer na strukturu zařízení, které přerušení požaduje, druhý argument byl získán z volání `bus_alloc_resource`, třetí parametr jsou příznaky, které určují, kdy bude přerušení zamaskováno: `INTR_TYPE_BIO` (přerušení bude maskováno při operacích s diskovými buffery), `INTR_TYPE_CAM` (přerušení bude maskováno při operacích se SCSI zařízeními), `INTR_TYPE_NET` (maskování při operacích se sítí), `INTR_TYPE_TTY` (maskování při operacích s terminálem nebo jinými znakovými zařízeními), `INTR_TYPE_MISC` (žádné maskování), `INTR_TYPE_FAST` (rychlá obsluha přerušení — při vyvolání přerušení se neprovádí maskování na řadiči, které je pomalé. Nicméně během běhu obslužné rutiny pak nemohou být zpracovávána jiná přerušení). Čtvrtý parametr je vlastní obslužná rutina, pátý argument, který se jí bude předávat, a na místo šestého parametru se uloží pointer na strukturu popisující přerušení, který se předá funkci pro rušení přerušení.

Mechanismus obsluhy přerušení funguje následovně: Systém má proměnné `cpl` a `ipending`. `cpl` obsahuje bitovou masku zamaskovaných přerušení. `ipending` obsahuje bitovou masku přerušení, která se vyskytla, ale nemohla být provedena, protože byla zamaskována. Při výskytu přerušení systém zamaskuje přerušení na řadiči. Pak zkontroluje, zda přerušení není zamaskované příslušným bitem v `cpl`. Pokud není, vykoná se obslužná rutina. Pokud je, nastaví se příslušný bit v `ipending` a ihned se vrátí.

Kód jádra může maskovat přerušení pomocí funkcí `unsigned splXXX()`. Funkce vrátí předchozí masku a novou masku omezí. `XXX` je třeba nahradit příslušným omezením — například `splbio()` zamaskuje všechna přerušení typu `INTR_TYPE_BIO`, `splnet()` zamaskuje všechna přerušení typu `INTR_TYPE_NET` a podobně. `void splx(unsigned x)` vrátí masku na původní hodnotu (`x` je hodnota, kterou vrátila příslušná funkce `splXXX`) a zkontroluje, zda některé nově odmaskované přerušení není nastaveno v proměnné `ipending`. Pokud ano, je obslužná rutina tohoto přerušení ihned zavolána. Ve FreeBSD se proces může zablokovat i se zamaskovanými přerušeními, nicméně po dobu zablokování procesu toto maskování neplatí. Při odblokování je obnovena maska přerušení, jaká byla ve stavu zablokování.

Systém maskování přerušení ve FreeBSD je lepší než v Linuxu — kód jádra maskuje pouze skupinu přerušení, která by mohla modifikovat struktury, k nimž přistupuje, nikoli všechna přerušení tak, jak je tomu na Linuxu. To přispívá k tomu, že FreeBSD má

menší latenci přerušení a umí lépe komunikovat se zařízeními, která potřebují okamžitou obsluhu.

V experimentálním FreeBSD 5 byl tento systém maskování přerušení odstraněn[2]. Ve FreeBSD 5 jsou přerušení zpracovávána přes *interrupt thready* — každé přerušení dostane kernel thread a při jeho výskytu se na tento kernel thread přepne (tohle vyvrací obecný princip o nepřepínání procesů uvnitř jádra). Interrupt thready mají i priority, což určuje priority jednotlivých přerušení. Interrupt thready se taktéž mohou zablokovat. Na druhou stranu, přepínání threadů je jistě pomalejší než přímé zavolání obslužné rutiny na zásobníku procesu, kde se vyskytla. FreeBSD 5 má i *fast interrupty*, které nemají kernel thread a jejich obslužná rutina se volá přímo. Fast interrupty se maskují na procesoru, zamaskováním všech přerušení. Vývojáři říkají, že `spl` zase zavedou ... je otázka, zda to skutečně udělají. Systém přerušení v experimentálním FreeBSD 5 mi připadá celkem nedodělaný — bude asi trvat dlouho, než z něj vzejde něco stabilního a než budou celý kód jádra a všechny ovladače přepsány, aby v nich nebyly race-conditiony.

Pro úplnost zde ještě popíšu systém přerušení, jaký používají Windows NT (a nejspíš i 2000 a XP). Procesor má proměnnou IRQL — ta určuje, jaká přerušení se mohou zpracovávat. Při registraci přerušení ovladač určí, na jaké IRQL toto přerušení bude běžet. Když se vyskytne přerušení, zjistí se, zda je IRQL přerušení větší než aktuální IRQL, na kterém daný procesor běží. Pokud ano, IRQL procesoru se zvětší, provede se obslužná rutina přerušení, a pak se IRQL zase zmenší na původní hodnotu. Pokud ne, obslužná rutina se odloží. Až bude IRQL procesoru sníženo, provedou se obslužné rutiny odložených přerušení s vyšší hodnotou IRQL. Maskování a odmaskování přerušení se provádí zvýšením a snížením IRQL na příslušnou hodnotu. Přerušení, která jsou krátká a potřebují být obsloužena v krátkém časovém intervalu, mají vysoká IRQL; přerušení, která trvají dlouho a na jejich latenci tolik nezáleží, mají nízká IRQL.

Mechanismus maskování přerušení na NT je lepší než na FreeBSD 4. NT při maskování určitého přerušení maskují i všechna pomalejší přerušení. Naproti tomu FreeBSD 4 maskuje pouze přerušení (nebo skupinu přerušení), které zamaskovat má. Představme si, že máme pomalé přerušení od disku, jehož obsluha trvá dlouho, a rychlé přerušení od zvukové karty, které musí být obslouženo okamžitě, jinak karta začne vydávat praskavé zvuky. Na FreeBSD 4 například může nastat situace: zamaskujeme přerušení od zvukové karty, přijde přerušení od disku, bude se vykonávat obslužná rutina přerušení od disku, přijde přerušení od zvukové karty, to však musí čekat, než skončí dlouhé přerušení od disku, a je obslouženo pozdě. Na NT tahle situace nastat nemůže, neboť při zvednutí IRQL na vysokou hodnotu, kterou má přerušení od zvukové karty, se zamaskují i všechna přerušení s nízkým IRQL. Mechanismus IRQL může garantovat určitou dobu odezvy pro realtimová zařízení.

[2] Tohle odstranění bylo dle mého názoru provedeno dost nešťastným způsobem — funkce `spl` byly prostě nahrazeny prázdnými funkcemi (viz soubor `sys/systm.h`) — to nejspíš povede k velkému množství race-conditions v ovladačích.

4.3. Softwarové přerušení

Softwarová přerušení fungují podobně jako hardwarová — s jediným rozdílem. Přerušení není vyvoláno hardwarovou událostí, ale samotným jádrem systému. Typickým použitím softwarových přerušení je zpracovávání síťových packetů — hardwarové přerušení přijme packet ze síťové karty. Kdyby zpracovávání packetu bylo prováděno rovnou v obsluze tohoto hardwarového přerušení, tak by po dobu zpracovávání bylo přerušení zamaskované. Zpracování packetu však může trvat dlouho a během této doby by nebylo možno přijímat další packety.

Proto je síťový subsystém dělán následovně — hardwarové přerušení přijme packet, uloží ho do fronty, vyvolá softwarové přerušení síťového stacku a skončí. Packet je dále zpracováván v softwarovém přerušení. Během obsluhy softwarového přerušení mohou přicházet další hardwarová přerušení a mohou být přijímány další packety.

Linux ve verzi 2.2 a nižší volal obsluhy softwarových přerušení po ukončení hardwarového přerušení. Ve verzi Linuxu 2.4 jsou softwarová přerušení obsluhována speciálním kernel threadem `ksoftirqd`. Nalezneme ho v souboru `kernel/softirq.c`. Ve víceprocesorovém systému běží několik těchto threadů — pro každý procesor jeden. U softwarových přerušení je třeba zajistit, aby nezahltila celý systém a nedošlo k zatuhnutí (například, když ze sítě chodí packety rychleji, než trvá jejich zpracování). V Linuxu 2.2 se to řešilo tak, že pokud bylo vyvoláno softwarové přerušení v době jeho vykonávání, systém obsluhu nového přerušení nevyvolal okamžitě, ale odložil ji až do dalšího hardwarového přerušení. V Linuxu 2.4 je ochrana proti zahlcení jednoduchá — kernel thread má v sobě funkci pro podmíněné schedulování.

Na FreeBSD 4 jsou softwarová přerušení zpracovávána stejně jako hardwarová (pomocí masek `cpl` a `ipending`). Na FreeBSD 5 jsou softwarová přerušení zpracovávána pomocí kernel threadů — opět stejně jako hardwarová přerušení.

4.4. Čekací fronty

Pokud chce proces čekat na nějakou událost, používá k tomu čekací fronty. V Linuxu čekací fronty nalezneme v souboru `include/linux/wait.h`. Čekací fronta je typu `wait_queue_head_t`. Když chce proces čekat na nějakou událost, zařadí se do příslušné fronty pomocí funkce `void sleep_on(wait_queue_head_t *queue)`. Po zavolání této funkce se proces zablokuje. Funkce `void wake_up(wait_queue_head_t *queue)` probudí všechny procesy, které na dané frontě čekají. Při čekání je též možno použít funkce `void sleep_on_interruptible(wait_queue_head_t *queue)`, která zajistí, že proces bude probuzen nejen funkcí `wake_up`, ale také příchodem signálu. Funkci `wake_up` je možno volat z kontextu přerušení; funkci `sleep_on` pochopitelně ne, protože způsobuje zablokování.

Používání funkce `sleep_on` je ve většině případů nevhodné. Pokud kód vypadá jako `if (nenastala_podmínka) sleep_on(čekací_fronta_na_podmínku)`, nastane problém v případě, kdy ona podmínka[3]nastane po `if`, ale před `sleep_on`. Pak bude proces čekat

[3] Podmínkami, na které se čeká ve frontách, jsou například natažení bloku dat z disku,

věčně ve frontě, ačkoli podmínka, na kterou čeká, je splněná. Aby se tomuto nežádoucímu tuhnutí zabránilo, proces se nejdřív umístí na frontu, pak otestuje platnost podmínky a pak teprve čeká.

Příklad takového čekání na podmínku (z funkce `wait_on_inode`, která čeká, než bude inoda načtená z disku):

```
DECLARE_WAITQUEUE(wait, current); /* deklarace struktury wait, která se bude umisťovat do fronty */
wait_queue_head_t *wq = i_waitq_head(inode); /* wq je fronta, na které se bude čekat */
repeat:  set_current_state(TASK_UNINTERRUPTIBLE); /* nastavení, že proces bude nepřerušitelný signálem */
if (inode->i_state & I_LOCK) { schedule(); /* dokud je inoda zamčená, zablokuj se a čekej */; goto repeat; }
remove_wait_queue(wq, &wait); /* odstranění struktury wait z fronty */;
current->state = TASK_RUNNING; /* odblokování procesu */
```

Tento způsob čekání je korektní: pokud je proces ve stavu `TASK_UNINTERRUPTIBLE`, funkce `schedule` provede zablokování procesu. Při odemčení inody jsou procesy z fronty vybrány a je jim nastaven stav na `TASK_RUNNING`. Pokud byla inoda odemčena po otestování `I_LOCK` a před zavoláním `schedule`, je proces nastaven do stavu `TASK_RUNNING` a volání `schedule` nezpůsobí zablokování.

Ve FreeBSD se k čekání používá funkce `tsleep` a k probuzení `wakeup`. Jejich funkce je stejná jako linuxové `sleep_on(_interruptible)` a `wake_up`. FreeBSD má i funkce `asleep`, která přidá proces do fronty, ale nechá ho běžet, a `await`, která čeká na událost registrovanou dříve pomocí `asleep`. Použití je zcela analogické jako na Linuxu — nejdřív zavolat `asleep`, pak otestovat podmínku a pak zavolat `await`, aby nedocházelo k nekonečnému čekání, pokud podmínka nastane rovnou po jejím otestování.

Nicméně implementace těchto funkcí je velmi mizerná — nevytváří se žádná fronta procesů jako na Linuxu a funkce `wakeup` prochází všechny procesy v systému, aby zjistila, který na danou událost čeká. Procházení všech procesů je celkem náročná operace a v tomto případě je to zcela zbytečné. Docela se divím, že to tak mohl nějaký programátor vůbec napsat. Ve FreeBSD 5 byla konečně napsána lepší implementace pomocí struktury `struct cv` a operací `cv_wait`, `cv_wait_sig` (čekání přerušitelné signálem), `cv_signal` (vzbudí jednoho) a `cv_broadcast` (vzbudí všechny). Tato implementace je ekvivalentní linuxovým wait queue. Bohužel se zatím používá pouze na několika málo místech — ve většině jádra je zatím neefektivní `tsleep/asleep/wakeup`.

4.5. Zamykání

Zamykání slouží k zajištění, aby se do kritické sekce nedostal více než jeden proces současně (pokud se tam nemají dostat ani obsluhy přerušení, je třeba kromě zamykání používat i maskování přerušení z předchozí kapitoly). Funkce na zamykání mohou způ-

stisk klávesy na terminálu, příchod dat na socket ze sítě, odswapování paměti a spousta dalších.

sobit zablokování procesu, a proto je není možno volat z obsluhy přerušení (výjimkou je FreeBSD 5 se svými interrupt thready).

Na Linuxu se k zamykání používají obyčejné semafory. Nalezneme je v souboru `include/asm/semaphore.h`. Na `struct semaphore` je možno provádět operace `up` a `down`. Existuje ještě operace `down_interruptible`, kdy čekání může být přerušeno signálem. Tyto funkce používají čekací fronty popsané v předchozí kapitole.

Na FreeBSD je implementace zámků výrazně komplikovanější. Zámek je popsán objemnou strukturou `struct lock` v souboru `sys_lock.h`. Na zámku se provádějí operace pomocí funkce `int lockmgr(struct lock *lock, unsigned int flags, struct simplelock *spinlock, struct proc *process);`. První parametr je zámek, poslední parametr proces provádějící operaci, třetí parametr je spinlock, který bude uvolněn (funkci spinlocků popíšu v následující kapitole o víceprocesorových systémech), a druhý parametr je operace, která se má provést. Zámek je možno zamknout pro zápis (`LK_EXCLUSIVE`) nebo pro čtení (`LK_SHARED`). Uvolnění zámku se dělá pomocí `LK_RELEASE`. Zámek může být zamčen pro čtení několika procesy současně. Pokud je zámek zamčen procesem pro zápis, pak žádný jiný proces nemá zámek zamčený pro čtení ani zápis. Funkce `lockmgr` taktéž umožňuje změnit typ zámku — čtení na zápis (`LK_UPGRADE`) nebo naopak (`LK_DOWNGRADE`). Pokud je použit příznak `LK_CANRECURSE`, zámky budou rekurzivní — t.j. tentýž proces bude moct několikrát zamknout tentýž zámek (a k odemčení bude potřeba tolik operací odemčení, kolikrát byl zámek zamknut). Funkce `lockmgr` taktéž kontroluje korektnost volání — vyhlásí panic, pokud proces uvolní zámek, který nevlastní, pokud proces zamkne jeden zámek vícekrát (a nenastaví `LK_CANRECURSE`) nebo pokud nesprávným způsobem mění typ zámku. Není prováděn test na deadlock, protože to by bylo příliš pomalé.

Implementace zámků ve FreeBSD poskytuje více možností než na Linuxu, ale není to výhoda. Zatímco na Linuxu má operace zamčení i odemčení dvě instrukce procesoru (na IA32), na FreeBSD se kvůli tomu volá celá komplikovaná funkce `lockmgr`, která z parametrů testuje, o jakou operaci se vlastně jedná, jaké má příznaky, zda daný proces zámek skutečně drží a kolikrát ho drží. Taková funkce je velmi pomalá. Kdyby měla být několikrát volána například při každém syscallu `read` pro zamykání a odemykání souboru a inody, bylo by to znát. Výsledek je ten, že se v jádře FreeBSD `lockmgr` příliš nepoužívá (používá se jen na místech, kde nejde o rychlost) a většina kódu si zamykání dělá sama, pomocí čekacích front. Taková komplikovaná implementace zámků má i další nevýhodu — umožňuje to lidem psát nečistý kód. Pokud někdo při navrhování nějakého subsystému potřebuje rekurzivní zamykání téhož zámku, svědčí to o tom, že kód a data špatně navrhl, a měl by místo používání komplikovaných zámků svůj návrh zjednodušit.

5. SMP

SMP znamená Symetric MultiProcessor a je to označení pro systémy s více procesory. Všechny procesory jsou si rovnocenné (vyjma startu systému, který je prováděn pouze jedním procesorem). Paměť je přístupná všem procesorům stejně. Je třeba si uvědomit, že současné procesory mění při provádění instrukcí jejich pořadí nebo složitější instrukce rozkládají na jednoduché mikroinstrukce, a proto je třeba počítat s tím, že ostatní procesory uvidí změny provedené jedním procesorem v libovolném pořadí. Pokud máme například deklarované proměnné `volatile int a = 0, b = 0;` a jeden procesor provede `a = 1; b = 2;`, může druhý procesor po určitý časový okamžik vidět `b == 2 && a == 0`[1]. Pokud oba procesory provedou `a++`, může být výsledná hodnota `a` 1 nebo 2 — například na Pentiu 2 se totiž instrukce `incl a` rozloží na mikroinstrukce „přečti hodnotu z paměti“, „zvětši hodnotu o 1“ a „ulož hodnotu do paměti“. Když se náhodou oba procesory sejdou tak, že budou současně číst, přičítat jedničku a ukládat do paměti, bude výsledná hodnota proměnné 1. Pokud nám záleží na pořadí, je třeba používat speciální atomické instrukce procesoru. U atomických instrukcí procesor zajistí, že jejich pořadí nebude měněno a že jiný procesor nebude moci přistupovat k paměťovým místům modifikovaným touto instrukcí. Kdyby ve výše uvedeném případě oba procesory provedly instrukci `lock;incl a`, bylo by zaručeno, že výsledná hodnota bude vždy 2.

Další problémy se SMP nastávají proto, že procesor může přehazovat vykonávání jednotlivých (mikro-)instrukcí. Může například načítat data dopředu, aniž zatím ví, zda je bude potřebovat, nebo může naopak zápisy opožďovat. IA32 garantuje, že ostatní procesory uvidí zápisy v tom pořadí, v jakém byly provedeny, nicméně na jiných architekturách není zaručeno ani to. Aby se efektům spojeným s přehazováním instrukcí dalo zabránit, byly vytvořeny speciální instrukce pro bariéry. `rmb()` je read-barrier a zaručuje, že žádné čtení nebude posunuto přes tuhle instrukci. `wmb()` je write-barrier a zaručuje, že žádný zápis nebude opožděn za tuto instrukci. `mb()` je bariéra pro čtení i zápis současně.

Při programování na SMP je třeba dávat zvláštní pozor na využití cachí procesoru. Každý procesor má svou vlastní L1 a L2 cache a tyto cache jsou mezi jednotlivými procesory synchronizovány pomocí metody MESI. MESI je zkratka složená z počátečních písmen názvů stavů, v jakých se řádka cache může nacházet: *modified*, *exclusive*, *shared* a *invalid*. Řádka je ve stavu *shared*, pokud obsah řádky odpovídá obsahu paměti a ostatní procesory mohou mít tutéž řádku ve své cachi. Stav *exclusive* znamená, že obsah řádky odpovídá obsahu paměti a žádný jiný procesor nemá řádku v cachi. Stav *modified* znamená, že řádka byla modifikována a musí být zapsána do paměti a žádný jiný procesor tuto řádku v cachi nemá. Procesor může řádku cache číst ve všech stavech vyjma *invalid*. Procesor může do řádky cache zapisovat ve stavech *modified* a *exclusive* (v takovém případě stav změní na *modified*). Pokud procesor chce zapisovat a řádka je ve stavu *shared*, musí nejdříve požádat ostatní procesory, aby tuto řádku ze svých cachí uvolnily (t.j. převedli ji do stavu *invalid*). To je pomalé, neboť to vyžaduje komunikaci po sběrnici. Každý procesor rovněž poslouchá všechny transakce na sběrnici — pokud jiný procesor

[1] Pokud deklarace proměnných není `volatile` může změnu pořadí přiřazení i čtení provádět už samotný kompilátor — nicméně proti přehazování instrukcí procesorem nás `volatile` neochrání.

čte řádku, která je ve stavu *exclusive*, změní stav na *shared*. Pokud jiný procesor čte řádku, která je ve stavu *modified*, procesor zruší transakci a zapíše svou modifikovanou řádku do paměti. To je opět velmi pomalé. Z toho plyne poučení pro programování: pokud budou dva procesory vykonávat kód, který modifikuje tutéž paměť, nebo pokud bude jeden procesor do paměti zapisovat a druhý z toho samého místa číst, vykonávání bude velmi pomalé. V extrémním případě to může být i několikanásobně pomalejší než na jednoprocesorovém systému, neboť na jednoprocesorovém systému by cache spoustu přístupů zachytila, zatímco na víceprocesorovém systému bude docházet k neustálému přelévání cachových řádek po sběrnici mezi oběma procesory a jejich zneplatňování.

Nyní popíšu, jakým způsobem byla na Linuxu a na FreeBSD podpora SMP implementována.

Obě jádra byla zpočátku navržena jako jednoprocesorová. Návrháři s možností více procesorů nepočítali. Kdyby takové jádro bylo puštěno na více procesorech současně, došlo by tam k velkému množství race-conditions (například už jen konstrukce `while (lock) sleep_on(&queue); lock = 1; ...` kritická sekce `...; lock = 0; wake_up(&queue)`, která se velmi často používala k zamykání, selže a do kritické sekce nám pustí oba procesory, pokud se sejdou v nevhodný okamžik). Nejjednodušší řešení takových problémů je takové, že do jádra pustíme nejvýše jeden procesor. Uživatelské procesy jsou vykonávány na všech procesorech současně, ale do jádra smí jen jeden. Pokud je nějaký proces v jádře a jiný proces na jiném procesoru chce vykonat nějaký syscall, musí počkat, než se první proces z jádra dostane. Taková implementace je velmi jednoduchá — stačí vzít jednoprocesorové jádro, do všech vstupů do jádra (t.j. syscall, interrupt, page-fault) přidat zámek — a už to může na víceprocesorovém stroji fungovat. Takhle to bylo implementováno v Linuxu 2.0 a FreeBSD 3. Problém této implementace je zcela evidentní — pokud procesy dělají hodně syscallů, tak pořád čekají na zámek, než se dostanou do jádra, a výkon se snižuje až na úroveň jednoprocesorového systému (nebo je dokonce ještě horší). Pro úlohy, které stráví většinu času v USER módu (například kompilace nebo různé výpočty), je taková implementace dostačující, nicméně pro úlohy, které volají často syscally (například síťový server), je naprosto nepoužitelná.

5.1. Spinlocky

V Linuxu 2.2 a FreeBSD 4 byla proto udělána lepší implementace SMP. Do jádra se již může dostat více procesorů najednou. K zamykání na krátkou dobu se používají `spinlocky`. Spinlock je jednoduchý zámek, který používá aktivní čekání. Pokud je spinlock zamknut a jiný procesor se ho pokusí znovu zamknout, bude tento procesor čekat v nekonečné smyčce, než první procesor spinlock uvolní. Pro zamčení spinlocku se používá atomická instrukce procesoru. Na Linuxu je spinlock deklarován ve struktuře `spinlock_t`. Pomocí makra `spin_lock(spinlock_t lock)` je možno spinlock zamknout. Pokud je spinlock již zamčen, čeká tato funkce v nekonečné smyčce. Spinlock se odemyká makrem `spin_unlock(spinlock_t lock)`. Když je jádro zkompilováno pro jednoprocesorový systém, spinlocky jsou prázdné struktury a tato makra nedělají nic. Pokud proces drží nějaký spinlock, nesmí se zablokovat. Při zablokování dochází k přepnutí procesu

a mohlo by se stát, že nový proces bude chtít zamknout tentýž spinlock a bude pak v nekonečné smyčce čekat na uvolnění, které nikdy nepřijde. Na Linuxu existují ještě rw-locky, což jsou spinlocky, které je možno zamykat pro čtení a pro zápis. Deklarují se ve struktuře `rwlock_t` a je možno na nich provádět operace `read_lock`, `read_unlock`, `write_lock` a `write_unlock`.

Přerušení na víceprocesorovém systému mohou přicházet na libovolný procesor. Linux 2.4 má možnost přiřadit určitému přerušení jeden konkrétní procesor. Výhodou je lepší využití cachí — data, se kterými manipuluje obslužná rutina přerušení, budou moci být v cachi procesoru a nebudou přelévána mezi více procesory. Určitou nevýhodou může být nevyvážená zátěž procesorů. Pokud máme dvouprocesorový stroj se dvěma stejně zatíženými síťovými kartami, je dobré přerušení od každé karty přiřadit na jeden procesor. Pokud máme jen jednu síťovou kartu, je dobré přerušení přiřadit jednomu určitému procesoru, ale jen do chvíle, než daný procesor zátěž TCP/IP stacku přestane zvládat. Pak je vhodnější přerušení nechat posílat na oba procesory, oba procesory nechat zpracovávat zátěž a smířit se s tím, že k přelévání dat mezi cachemi bude docházet.

K synchronizaci s obsluhami přerušení se rovněž používají spinlocky. Rozdíl je v tom, že před vlastním zamčením spinlocku musí kód jádra zakázat přerušení na procesoru, na kterém běží. Kdyby kód jádra zamknul spinlock bez zakázání přerušení, mohlo by se stát, že přerušení přijde a bude chtít zamknout tentýž spinlock — to by ovšem znamenalo nekonečné čekání a zatuhnutí. K zamykání a odemykání je možno použít makra `spin_lock_irq(spinlock_t spinlock)` a `spin_unlock_irq(spinlock_t spinlock)`, která na procesoru, na kterém běží, zakážou přerušení před vlastním zamčením a povolí je po odemčení, nebo makra `spin_lock_irqsave(spinlock_t spinlock, unsigned long flags)` a `spin_lock_irqrestore(spinlock_t spinlock, unsigned long flags)`, která před zakázáním přerušení uloží stav zakázání do proměnné flags a po odemčení spinlocku tento stav obnoví[2]. Pokud je jádro zkompilováno jako jednoprocesorové, `spin_lock_irq` funguje jako `cli`, `spin_unlock_irq` jako `sti`, `spin_lock_irqsave` jako `save_flags; cli` a `spin_unlock_irqsave` jako `restore_flags`. Zakázání přerušení pomocí `cli` a `sti` se v Linuxu už téměř nepoužívá. Zakázání přerušení na všech procesorech je moc pomalé a zakázání přerušení na jednom procesoru nepomůže. K synchronizaci s obslužnými rutinami přerušení se používá zakázání přerušení na jednom procesoru a spinlock (pomocí maker `spin_lock_irq` a `spin_unlock_irq`). Obslužná rutina přerušení pak vezme spinlock pomocí `spin_lock` a `spin_unlock`. U kódu, o kterém nevíme, zda bude volán se zakázanými, nebo povolenými přerušeními, je třeba použít `spin_lock_irqsave` a `spin_lock_irqrestore`. Tato metoda zamykání je v Linuxu používána téměř všude. Je zajištěno, že přerušení nepřijde na procesor, který zákaz provedl. Pokud přerušení přijde na jiný procesor, bude jeho obslužná rutina čekat ve spinlocku, než skončí kritická sekce.

Ve FreeBSD 4 existují také spinlocky, ale nazývají se simple locky. Jsou deklarované ve struktuře `struct simple_lock` a je možno na nich provádět funkce `simple_lock` a `simple_unlock`, které jsou funkčně ekvivalentní linuxovým `spin_lock` a `spin_unlock`. Na FreeBSD to jsou skutečné funkce — ne makra ani inline funkce — proto jsou mírně

[2] Jedná se o makra preprocesoru a nikoli funkce, proto `spin_lock_irqsave` do proměnné `flags` zapisuje, ačkoli není tato proměnná předávána jako pointer.

pomalejší.

Na FreeBSD 5 simple locky vymizely a objevila se nová struktura `struct mtx`. Tuto strukturu je možno používat jak pro spinlocky, tak pro sleep locky, které proces zablokují, pokud je zámek zamčený. Pokud je tato struktura použita jako blokující zámek, k zamykání a odemykání se používají funkce `mtx_lock` a `mtx_unlock`. Pokud je použita jako spinlock, používají se `mtx_lock_spin` a `mtx_unlock_spin`. Tyto funkce pro spinlocky zakazují a povolují přerušení jako linuxové `spin_lock_irqsave` a `spin_lock_irqrestore`. Funkce jsou zčásti inlinované a implementace blokujících zámků je výrazně lepší než `lockmgr` (který byl naprosto nepoužitelný) nicméně jednoduchosti a rychlosti linuxových zámků stále nedosahují. Zejména možnost rekursivních zámků (pokud je při inicializaci nastaven příslušný příznak) je dle mého názoru zcela zbytečná. Spinlocky navíc nejsou plně inlinované — volají pořád funkci na zakázání přerušení a uložení stavu. FreeBSD 5 má i blokující read-write locky ve struktuře `struct sx`. Ve FreeBSD 5 byl napsán speciální debuggovací kód nazvaný witness, který kontroluje všechny operace se zámky a ohlásí chybu, pokud je porušeno pořadí braní zámků (neboť při nedodržení pořadí hrozí deadlock) nebo pokud proces provede nedovolenou operaci — například se zablokuje a drží spinlock. Výhoda tohoto kódu je taková, že přímo ohlásí jméno souboru i řádku, kde k takové operaci došlo. Pokud je při kompilaci jádra witness povolen, způsobuje to samozřejmě velké zpomalení, neboť je třeba udržovat seznam držených zámků pro každý proces.

5.2. Big kernel lock

Při zavádění spinlocků bylo potřeba velké části kódu přepsat. To nebylo možno provést naráz pro celé jádro. Aby mohly být nadále používány části kódu napsané pro původní jednoprocesorové jádro, byl zaveden big kernel lock. Big kernel lock je spinlock, který má speciální vlastnost — proces držící tento spinlock se může zablokovat. Při zablokování procesu dojde k uvolnění big kernel locku a při opětovném probuzení procesu je lock opět zamčen. Kód napsaný pro původní jednoprocesorová jádra může být nadále používán — musí však běžet s big kernel lockem zamčeným.

Na Linuxu big kernel lock zamykáme a odemykáme pomocí funkcí `lock_kernel()` a `unlock_kernel()`. Na FreeBSD 4 se používají funkce `get_mplock()` a `rel_mplock()`, na FreeBSD 5 `mtx_lock(&Giant)` a `mtx_unlock(&Giant)`. Na všech systémech je zámek rekursivní — na Linuxu je to asi jediný rekursivní zámek, který tam existuje. Na FreeBSD 4 navíc provedení `get_mplock` způsobí, že interrupty budou přicházet jen na ten jeden procesor, který `get_mplock` provedl. To zajistí, že synchronizace přerušení pomocí `spl` funkcí bude fungovat.

Kód běžící s big kernel lockem je nežádoucí, protože ho není možno paralelně vykonávat na více procesorech. Proto je snaha vývojářů tento big kernel lock odstraňovat. Na Linuxu 2.2 se nacházel na mnoha místech — v celém filesystému, swapperu a ve všech síťových funkcích. Na Linuxu 2.4 byl ve spoustě běžně vykonávaného kódu odstraněn — již je možné číst a zapisovat do souborů, swapovat stránky a provádět síťové operace, aniž by byl big kernel lock sebrán. Tyto operace je tedy možno provádět paralelně na několika procesorech, což vede k výrazně lepšímu využití víceprocesorového systému. FreeBSD je

na tom se SMP paralelismem výrazně hůř. FreeBSD 4 sice synchronizační primitiva pro SMP má, nicméně jsou velmi málo využívána a vyjma osmi jednoduchých sycallů běží všechny s big kernel lockem. Ještě horší než nedostatečný SMP paralelismus je fakt, že scheduler FreeBSD 4 obsahuje bug, který se projevuje jen na SMP a způsobuje, že pokud jsou všechny procesory vytížené, mají interaktivní procesy až několikasekundovou odezvu. Na takovém systému se pod zátěží v podstatě vůbec nedá pracovat. Na FreeBSD 5 SMP paralelismus pokročil; syscally již neberou big kernel lock hned při vstupu, v jádře se již začínají objevovat spinlocky, nicméně s big kernel lockem běží celé síťování, filesystém i správa paměti. Řekl bych, že FreeBSD 5 je tak asi úrovni Linuxu 2.2.

5.3. Chyby v SMP

SMP s sebou přináší i další problém — a to jsou notoricky se vyskytující chyby v jádrech. Při psaní jádra pro jednoprocesorový systém člověk moc často chybu neudělá. K přepnutí procesu tam může dojít jen na stanovených místech a na těch je možno ohlídat, zda jiný proces nezměnil nebo neuvolnil struktury, se kterými právě pracujeme. Na SMP je situace mnohem nepřehlednější, neboť v jádře se může nacházet více procesorů současně a k synchronizaci je třeba spinlocky důsledně využívat. Nejhorší na tom je, že bugy v SMP synchronizaci se projevují velmi zřídka — pokud například programátor zapomene použít spinlock, systém zdánlivě běží v pořádku a vytuhne nebo spadne třeba jednou za měsíc, až se oba procesory náhodou zkříží v kritické sekci. To se samozřejmě ladí mnohem hůře, než kdyby se pád projevoval denně. Chyby v SMP zamykání v podstatě vůbec debuggovat nejdou a nezbývá, než si celý kód přečíst a ujišťovat se, že je zamykání správné.

Například v jádře Linuxu 2.4.18 byly opraveny SMP race-conditiony při alokaci PID procesů, v implementaci syscallu `dnotify` a v netfilteru. Já jsem našel SMP bug v Linuxu 2.4.18 a 2.5.23 ve funkci `mark_inode_dirty`, který mohl způsobit, že inoda nebude zapsána na filesystém, pokud ji jeden procesor modifikuje a druhý zapisuje (bug spočíval v nerespektování možnosti, že procesor přehazuje instrukce, a nepoužití bariéry). Jak vidno, nejedná se tedy o nepodstatné drobnosti, které se běžným uživatelům neprojevují. Nejhorší vlastností SMP race-conditionů je, že se projevují velmi zřídka.

V Linuxu 2.0 a 2.2 je zase bug při přístupu k tabulce stránek. Pokud na jednom procesoru běží prohledávání tabulek stránek při vyswapovávání na disk a na druhém procesoru běží proces, jehož tabulka stránek je prohledávána, může nastat následující situace: první procesor přečte položku tabulky stránek a zjistí, že má shozené příznaky ACCESED a DIRTY a že tedy může být odstraněna. Druhý procesor zapíše do oné stránky, čímž nastaví příznaky ACCESSED i DIRTY. První procesor přepíše položku na neplatnou (neboť tam dříve viděl bity ACCESSED a DIRTY shozeny) a odstraní stránku z paměti. Druhý případ téhle race-conditiony je takový, že na počátku má položka tabulky stránek nastaven bit ACCESSED a shozen bit DIRTY. První procesor přečte položku tabulky stránek a rozhodne se, že příznak ACCESSED zruší a stránku nechá v paměti. Druhý procesor zapíše do stránky, čímž nastaví DIRTY. První procesor přemaže položku tabulky stránek původní položkou se smazaným příznakem ACCESSED, čímž bohužel smaže i právě nastavený příznak DIRTY. Při přístupu k tabulce stránek na víceprocesorovém sys-

tému je třeba používat atomické instrukce procesoru. Tento bug projevuje jen u stránek namapovaných ze souboru při použití příznaků `MAP_SHARED` a `PROT_WRITE`. Naštěstí se neprojevuje u naalokovaných stránek nebo stránek namapovaných s `MAP_PRIVATE`. V jádrech 2.4 je opraven, vývojáři ho z mně neznámých důvodů odmítají opravit ve starších jádrech 2.2 a 2.0.

V Linuxu 2.2 je spousta dalších neopravených SMP bugů, například teď jsem v 2.2.21 našel špatnou implementaci funkcí `wait_on_buffer` a `wait_on_inode`, která může vést k zatuhnutí a nekonečnému čekání na inodu nebo buffer, který je již načten (nastane v situace, kdy jeden procesor současně dokončuje načtení a druhý začíná čekat — jedná se rovněž o nerespektování možnosti přehazování instrukcí). V jádrech 2.4 jsou tyto bugy opraveny.

Závěr: Podpora SMP ve FreeBSD je zcela nedostatečná; kvůli onomu bugu v scheduleru nedoporučuji nikomu FreeBSD 4 na SMP používat. FreeBSD 5 je sice lepší, nicméně to je experimentální verze, která by se na produktivních systémech neměla používat vůbec. V Linuxu 2.4 je podpora SMP kvalitní, nicméně to s sebou přináší riziko bugovitosti, takže bych nedoporučoval Linux 2.4 na SMP používat, pokud je potřeba absolutní spolehlivost. Pokud nebudeme mapovat soubory do paměti a zapisovat do nich, tak nejstabilnější na SMP bude asi jádro Linuxu 2.0, neboť v něm nehrozí SMP race-conditiony. Nicméně takové jádro je zcela neefektivní pro servery nebo jiné úlohy, které volají hodně syscallů.

5.4. Preemptivní jádro Linuxu

S podporou SMP v Linuxu souvisí i další vlastnost — možnost preemptivního přepínání procesů uvnitř jádra. Jádro bylo z větší části přepsáno tak, aby se do něj mohlo dostat více procesorů, a tyto procesory se synchronizují pomocí spinlocků. Někoho tedy napadlo, že by spinlocky šly využít i k synchronizaci mezi jednotlivými procesy a ty by pak bylo možno přepínat kdykoli.

Možnost preemptivního přepínání procesů existuje pouze v experimentálních jádrech Linuxu 2.5. Implicitně je vypnutá; musí se zapnout při konfiguraci jádra. Funguje následovně: spinlocky se chovají jako blokující semafory a procesy se přepínají kdykoli v jádře. Většina kódu napsaného pro SMP s nepreemptivním jádrem tak může fungovat i s preemptivním přepínáním procesů v jádře na jedno- nebo víceprocesorovém stroji. Problém nastane s kódem, který používal CPU-local storage (to je malá oblast paměti vyhrazená pro každý procesor). S nepreemptivním přepínáním procesů bylo možno, aby kód využíval CPU-local storage bez braní jakéhokoli spinlocku nebo zámku (pouze se během práce s CPU-local storage nesměl zablokovat), naproti tomu při preemptivním přepínání procesů v takovém kódu vznikne race-condition. Je potřeba všechny takové kusy kódu najít a přepsat pro použití preemptivního jádra. To nebude snadné. Proto preempce zatím není považována za stabilní.

Preemptivní jádro vede k rychlejšímu probuzení procesů, nicméně garantovanou dobu probuzení nám nedává. Preempce se totiž vypíná, pokud nějaký proces vezme big kernel lock pomocí `lock_kernel`. A takových míst se stále ještě vyskytuje mnoho, i když ubývají. Preempce také může vést k určitému zpomalení na jednoprocesorových systémech, neboť spinlocky, které bez preempce nedělaly nic, se nyní začnou chovat jako semafory.

6. Alokace paměti v jádře

V této kapitole popíšu algoritmy, jaké se používají k přidělování paměti samotnému jádru systému. Nebude zde popsáno přidělování paměti uživatelským procesům a swapování — to bude náležet až do některé další kapitoly.

Linux i FreeBSD byly navrženy tak, že stránky alokované jádrem jsou neswapovatelné. Struktury jádra zabírají pouze malou část paměti, a proto jejich neswapovatelnost nevadí. Existují i systémy, které mají některé struktury jádra swapovatelné — například AIX nebo Windows NT — nicméně to komplikuje programování v jádře a přináší to s sebou problémy. Proces se může zablokovat kdykoli při přístupu na swapovatelnou paměť, proto je nutné v tomto místě počítat s možným přepnutím procesu. Není tedy možno držet spinlock a přistupovat na swapovatelnou paměť.

Základní funkcí systému je operace alokace stránky. Každá fyzická stránka paměti je popsána strukturou `struct page` na Linuxu a `struct vm_page` na FreeBSD. Pole těchto struktur je staticky alokováno při startu systému. Jeho velikost je určená podle množství paměti. Požadavky na stránky se v zásadě dělí na dvě skupiny — na blokující a neblokující (též zvané atomické) alokace paměti. Při blokujícím požadavku je možné, že proces se zablokuje a počká, než swapper nějaké stránky odswapuje na disk, nebo než uvolní stránky z cache nebo diskových bufferů. Při atomickém požadavku se proces zablokovat nesmí, a pokud dojde paměť, alokační funkce vrátí `NULL`. Protože systém za běhu používá skoro celou paměť pro uživatelské procesy a jako diskovou cache, je v něm za běhu poměrně málo volných stránek a k selhání atomické alokace může dojít v podstatě kdykoli. Platí obecný princip: při blokující alokaci je garantováno, že proces paměť dostane (na Linuxu se může stát, že blokující alokace vrátí `NULL`, ale dochází k tomu jen v extrémním případě, kdy jádro zabralo celou paměť; na FreeBSD blokující alokace neselže nikdy, a pokud jádro zabere celou paměť, je zastaveno na `panic`). Naproti tomu atomická alokace může kdykoli selhat a kód se s tím musí vypořádat bez nějaké ztráty dat nebo narušení funkčnosti. Atomická alokace se používá v přerušení, neboť obslužná rutina přerušení se zablokovat nesmí. Typickým použitím atomické alokace je alokace paměti pro packety přicházející ze sítě. To se dělá v přerušení od ovladače síťové karty a selhání atomické alokace zde nevadí, neboť protokoly vyšších vrstev se se ztrátou packetu musejí umět vypořádat.

K zajištění dobré funkčnosti systému je třeba mít nějaký limit pro atomické alokace. Pokud je množství volných stránek menší než tento limit, je paměť přidělována už jen atomickým alokacím a blokující alokace musejí počkat, než se stránky odswapují a volná paměť stoupne nad limit. Kdyby takový limit neexistoval, mohlo by se například stát, že proces neustále alokující stránky by zablokoval celé síťování, neboť by okamžitě zabral jakoukoli uvolněnou stránku, na obslužnou rutinu přerušení síťové karty by už nezbylo nic a každý packet by byl zahozen.

6.1. Buddy alokátor na Linuxu

Na Linuxu slouží k alokaci stránek funkce `struct page *alloc_pages(unsigned int gfp_mask, unsigned int order)`. První parametr je konstanta `GFP_xxx` nebo

maska složená z konstant `__GPF_xxx`. `GPF_ATOMIC` určuje, že jde o atomickou alokaci, `GFP_KERNEL` znamená, že jde o běžnou blokující alokaci. `GPF_USER` se používá pro alokaci stránek, o které požádají uživatelské procesy — má stejný význam jako `GFP_KERNEL` až na to, že pokud je zaplněná celá paměť i swap, je limit pro `GFP_USER` vyšší než pro `GPF_KERNEL`. Je to proto, aby, když uživatelské procesy spotřebují celou paměť, ještě nějaká paměť zbyla jádru. `GPF_NOFS` a `GPF_NOIO` znamená, že alokátor nesmí zavolat filesystém nebo blokový io-systém, aby uvolnil paměť, protože by mohlo dojít k deadlocku (použití těchto dvou příznaků mi nepřipadá moc čisté — alokátor sice nemůže zavolat filesystém nebo io-systém, nicméně může klidně čekat, než swapovací proces kswapd nějakou paměť uvolní — a kswapd volá filesystém i io-systém. Jak uvidíme později, ono je to s těmi deadlocky docela zamotané a ne moc dobře vyřešené). Další příznaky určují, zda paměť musí ležet v dolních 16M paměti, aby byla použitelná pro ISA DMA, nebo zda může ležet v nenamapované high-memory zóně, o které bude řeč později.

Druhý parametr funkce `alloc_pages` je dvojkový logaritmus počtu stránek, který se má alokovat. Je-li to nula, alokuje se jedna stránka, je-li to 1 či 2, alokují se dvě či čtyři stránky a tak dále. Alokované stránky jsou v jednom souvislém bloku. Linux používá k alokaci buddy alokátor, který drží pro jednotlivé mocniny dvojky seznamy bloků příslušného počtu volných stránek. Je držen seznam všech volných nespárovatelných stránek. Dále je držen seznam všech volných dvojic stránek (dvojice stránek musí být zarovnaná — t.j. dolní stránka má pořadí, které je dělitelné dvěma), pak je držen seznam všech volných čtveřic stránek (dolní stránka čtveřice má pořadí dělitelné čtyřmi) a tak dále i pro vyšší mocniny dvojky. Při požadavku o alokaci určitého počtu stránek se alokátor nejdříve podívá do seznamu odpovídajícího danému počtu. Pokud tam blok stránek nalezne, tak ho přidělí. Pokud ho tam nenalezne, podívá se do dalšího seznamu (obsahujícího bloky dvojnásobné velikosti), a když tam nalezne blok, rozdělí ho na poloviny. Jednu polovinu uloží do nižšího seznamu bloků poloviční délky a druhou polovinu vrátí. Pokud nenalezne blok ani tam, podívá se do dalšího seznamu a tak dále, až narazí na seznam největších bloků. Pokud nenalezne volný blok ani tam, alokace selže. Při uvolnění bloku stránek se kontroluje, zda onen blok nemá volný blok v páru, a pokud ano, oba bloky se spojí a přidají se do vyššího seznamu. Pokud nový spojený blok má opět volný blok v páru, spojí se i tyhle, a tak dále.

Buddy alokátor s sebou přináší problém — a tím je fragmentace paměti. Pokud není spotřebovaná celá paměť, je garantováno, že se podaří alokovat jednu stránku; bohužel už není garantováno, že se podaří alokovat více souvislých stránek. Na starších jádrech 2.4 platilo, že alokace mohla kdykoli selhat, pokud parametr `order` byl větší než nula. Alokace dvou souvislých stránek je na IA32 potřeba k vytvoření procesu, proto se tam mohlo kdykoli stát, že vytvoření procesu selže a vrátí chybu `ENOMEM`. V jádře 2.4.18 je to napsáno tak, že pokud je `order <= 3` a alokace selže (a nejedná se o atomickou alokaci), bude se v cyklu neustále volat funkce na uvolnění nebo swapování stránek, doufajíc, že se blok potřebné velikosti podaří vytvořit (v dřívějších jádrech tento cyklus probíhal jen pro `order == 0`). Nepovede to sice k náhodnému selhání vytváření procesu jako dřív, nicméně při velmi nevhodném umístění stránek a nedostatku swapu může dojít k nekonečnému cyklu. Například pokud je alokována polovina paměti a stránky jsou tak nešikovně umístěny, že volné stránky netvoří žádný pár, nepodaří se vytvořit nový proces a tahle operace bude cyklit v nekonečné smyčce. Nicméně pravděpodobnost, že se to stane, je

velmi malá. Pokud je navíc dostatek swapovacího místa, budou stránky swapovány tak dlouho, dokud se blok potřebné velikosti nevytvoří.

6.2. Barvení stránek na FreeBSD

FreeBSD nemá buddy alokátor. K alokaci jedné stránky se používá funkce `struct *vm_page *vm_page_alloc(vm_object_t object, vm_pindex_t index, int page_req)`. První parametr je objekt, do kterého se má stránka uložit, a druhý je index v tomto objektu (vm_objekty budou popsány později; pro alokaci stránek pro jádro se používá objekt jádra `kernel_object`). Třetí parametr určuje prioritu alokace — nejnižší pro uživatelský proces, vyšší pro systém, nejvyšší pro interrupt. Tato funkce se nikdy neblokuje. Pokud paměť není nebo pokud je množství volné paměti pod kvótou pro danou prioritu alokace, vrátí `NULL`. Pokud vrátí `NULL`, může se kód, který ji volal, zablokovat pomocí makra `VM_WAIT;` a čekat, než swapper nějaké stránky odswapuje a nějaká paměť bude dostupná. Po probuzení z `VM_WAIT` je třeba opět zkusit alokaci pomocí `vm_page_alloc`.

Při alokaci používá FreeBSD algoritmus barvení stránek. Barvení stránek slouží k optimalizaci použití L2 cache. Pro pochopení algoritmu nejdříve popíši, jakým způsobem cache fungují. Nejmenší jednotka, s jakou cache pracuje, je *řádka*. Například na Pentiu až Pentiu 3 má řádka velikost 32 bytů, na Pentiu 4 má velikost 64 bytů. Ačkoli se říká, že cache je asociativní paměť, není to pravda. Udělat plně asociativní cache není technicky možné, proto má cache adresy. Dolních 5 nebo 6 bitů v adrese je pozice uvnitř řádky, dalších několik bitů adresy je adresa v cachi. Cache pracuje jako obyčejná paměť — těchto několik bitů se použije jako paměťová adresa a podle ní se v cachi najde příslušné místo, na kterém je uloženo několik řádek (typicky 1, 2 nebo 4 — podle toho se cache nazývá jednocestně, dvoucestně nebo čtyřcestně asociativní). Požadovaná celá adresa se porovná s celými adresami těchto několika řádek, a pokud je jedna z nich shodná, prohlásí se hodnota v cachi za nalezenou. Například Pentium 2 má L1 cache čtyřcestně asociativní o velikosti 16kB s délkou řádky 32B. Znamená to tedy, že dolních 5 bitů adresy je offset uvnitř řádky. Dalších 7 bitů je adresa v cachi (spočteno jako $log_2(16kB/32B/4)$). Na jedné adrese v cachi se nacházejí čtyři řádky. U Pentia 2 je L2 cache čtyřcestně asociativní, má velikost 512kB a délku řádky 32B. To znamená, že 5 bitů adresy je offset v řádce a 12 dalších bitů je adresa v cachi.

Cache pracuje s fyzickými adresami[1]. Pokud je například cache čtyřcestně asociativní a uživatelský program alokuje pět souvislých po sobě jdoucích stránek, může se stát, že stránky náhodou padnou na takové fyzické adresy, že všech pět stránek bude mít stejnou adresu v cachi. Pokud pak program bude tyto stránky procházet, nepodaří se je do cache dostat, neboť na jedné adrese čtyřcestné cache mohou být pouze čtyři řádky. I když má L2 cache velkou velikost 512kB, je možné, že se pět nevhodně umístěných stránek o celkové velikosti 20kB do téhle cache nevejde.

Aby k tomuto jevu nedocházelo, používá FreeBSD barvení stránek. Každá stránka

[1] Existují i architektury — například Sparc64 — kde cache pracuje s virtuálními adresami. Je s tím velké množství problémů, pokud dvě virtuální stránky ukazují na jednu stránku fyzickou.

má barvu, což je adresa dat v cachi. Například pokud máme čtyřcestnou cache o velikosti 512kB a velikost stránky 4kB, stránky mají 32 barev. Alokátor stránek ve FreeBSD se snaží přidělovat stránky takové barvy, která odpovídá virtuální adrese, na níž bude stránka namapována. Pokud například program alokuje pět stránek na virtuálních adresách 0, 4096, 8192, 12288 a 16384, budou tyto stránky mít barvy 0, 1, 2, 3, 4. Touto strategií se zabrání výše zmiňovanému jevu, kdy několik stránek má stejnou barvu (čili stejnou adresu v cachi) a tyto stránky se pak do cache nevejdou. Implementace barvení stránek je jednoduchá — pro každou barvu existuje fronta volných stránek, které onu barvu mají. Při požadavku o stránku se nejprve bere stránka z fronty s požadovanou barvou. Pokud je fronta prázdná, vezme se stránka z fronty s nejvzdálenější barvou. Je-li tato fronta taktéž prázdná, procházejí se ostatní fronty. Parametr `index` funkce `vm_page_alloc` se používá k určení požadované barvy stránky.

Otázka, zda barvení stránek pomáhá, nebo nepomáhá zvýšit rychlost běhu programů, je sporná. Tvůrci Linuxu tvrdí, že barvení zbytečně zesložiťuje jádro a jeho efekt je zanedbatelný, a proto ho nechtějí implementovat. Protože tvůrci FreeBSD barvení implementovali, tvrdí, že barvení pomáhá při zvýšení rychlosti. Realita je asi taková, že:

- Pokud program alokuje několikrát méně paměti, než je velikost L2 cache, pak pokud barvení stránek nebude použito, je malá pravděpodobnost, že bude mít spousta stránek stejnou barvu. Proto v takovém případě nemá barvení velký vliv.
- Pokud program alokuje několikrát více paměti, než je velikost L2 cache, a náhodně na tuto paměť přistupuje, tak se stránky do cache stejně nevejdou a vůbec nezáleží na tom, zda je barvení použito, nebo ne.
- Pokud program alokuje množství paměti srovnatelné s velikostí L2 cache, má barvení stránek význam. Náhodným přidělováním stránek bez barvení se nedosáhne rovnoměrného rozdělení barev a pokrytí celé cache. Naproti tomu barvení zajistí, že cache bude rovnoměrně pokrytá a že program dostane přesně takové množství stránek se stejnou barvou, jaká je asociativita cache. Díky tomu se celá datová oblast programu vejde do cache.

Závěr: Barvení stránek pomáhá v některých speciálních případech (například různé vědecké výpočty pracující s bloky dat přesně o velikosti cache), nicméně v běžných programech se příliš neprojeví.

6.3. Mapování stránek v jádře

V předchozí kapitole jsme si popsali alokaci stránek a nyní nastává otázka, jak k nim přistupovat. V současných systémech se množství fyzické paměti blíží velikosti virtuálního adresního prostoru nebo ji i překračuje. Procesory Pentium mohly mít maximálně 4G fyzické paměti a 4G virtuální paměti. Pentium Pro může adresovat až 64G fyzické paměti a 4G virtuální paměti. Pokud má systém více fyzické paměti než virtuální, nemohou být všechny stránky současně namapované a je třeba řešit situace, jak stránky mapovat. V této kapitole se budeme zabývat procesory IA32. U 64bitových procesorů je situace výrazně lepší — tyto procesory mohou adresovat mnohem větší množství virtuální paměti, a tak zde nehrozí nebezpečí, že by fyzické paměti bylo více než virtuální.

Na Linuxu 2.2 a nižších je třeba mít vždy celou fyzickou paměť namapovanou do

adresního prostoru jádra. Jádro má 1G virtuálních adres, uživatelské programy mají 3G. Z čehož plyne, že takové jádro může používat maximálně 1G paměti. Jádro je možno upravit tak, aby jeho adresní prostor byl 2G a uživatelské programy dostaly také 2G; s touto úpravou tedy je možno jádro používat na strojích s maximálně 2G paměti. Toto řešení není dostatečné, proto byl způsob mapování paměti v jádrech 2.4 změněn.

Linux 2.4 dělí paměť na zóny. Typicky máme tři zóny — zónu použitelnou pro ISA DMA o velikosti 16M (tato zóna existovala i na předchozích jádrech), zónu přímo namapovaných stránek, která má velikost 1G, a highmem zónu nenamapovaných stránek, která obsahuje všechny zbývající stránky nad hranicí 1G až do potenciálního limitu 64G na Pentiu Pro. Příznak `__GFP_DMA` určuje, že alokace musí být z ISA DMA zóny. Příznak `__GFP_HIGHMEM` určuje, že alokace může být z highmem zóny. Pokud není ani jeden příznak uveden, alokuje se z přímo namapované zóny, a pokud je ta prázdná, tak z ISA DMA zóny. Paměť o fyzických adresách 0 až 1G je permanentně namapovaná na virtuální adresy 3G až 4G (nejedná se přesně o 1G, je to trochu méně, aby v jádře ještě nějaké virtuální adresy zbyly). Z této paměti se alokují všechny struktury jádra. Je totiž potřeba, aby struktury jádra byly vždy dostupné. Z highmem zóny se alokují stránky uživatelských procesů a stránky diskové cache. U těchto stránek není potřeba, aby byly vždy přístupné jádru. Pokud jádro potřebuje přistupovat na nenamapovanou stránku, musí použít funkci `void *kmap(struct page *)` pro její namapování do virtuálního adresního prostoru jádra a `void kunmap(struct page *)` pro odmapování. Tyto funkce udržují cache 512 nebo 1024 naposledy namapovaných stránek, pokud stránka je už namapovaná v cachi, vrátí adresu, pokud není namapovaná, tak odmapují nějakou stránku z cache a požadovanou stránku namapují, pokud jsou všechny stránky v cachi právě používané, proces se zablokuje a čeká, než se nějaké mapování neuvolní. Toto mapování se provádí například při nulování čerstvě alokovaných uživatelských stránek nebo při kopírování stránek po provedení syscallu `fork`.

Při přechodu na nový způsob mapování stránek v jádrech 2.4 vznikl problém se starými ovladači zařízení. Například ovladače disku dostávají virtuální adresu dat a vyžadují, aby tato adresa byla namapovaná. Nejsou tedy schopny dělat I/O do highmem zóny. Řešením jsou bounce-buffery — pokud se má dělat I/O z nebo do highmem zóny, alokuje se dočasný bounce-buffer v namapovaná zóně, do něho se udělá I/O a data se kopírují. Pokud se provádí zápis, data se kopírují před zápisem do bounce-bufferu, pokud se provádí čtení, data se kopírují po čtení z bounce-bufferu. Problém je vyřešen, ovšem s nepříjemným důsledkem — data se občas kopírují, ačkoli by se kopírovat nemusela. Například PCI IDE DMA může pracovat s adresami až do 4G. Nicméně pokud děláme I/O v rozsahu 1G až 4G, musíme použít bounce-buffer, protože ovladač IDE neumí pracovat s nenamapovanými adresami. Použití bounce-bufferu a kopírování dat zpomaluje přístup na disk. V experimentálních jádrech 2.5 je tento problém již vyřešen — ovladače IDE a SCSI disků byly modifikovány tak, aby mohly pracovat s nenamapovanými buffery, a bouncování se používá opravdu jen tehdy, když je nutné — tedy při práci s daty nad 4G, pokud architektura neumožňuje mapování adresního prostoru PCI, které bude popsáno v následující kapitole.

FreeBSD funguje zcela jinak. Implicitně nemá namapovanou žádnou paměť a stránky jádra mapuje do virtuální paměti po alokaci. Každá stránka je mapovaná zvlášť. FreeBSD nemá a ani nepotřebuje buddy alokátor. Pokud pomocí je funkce `malloc` požadována

alokace bloku většího než stránka, alokuje se několik nesouvislých stránek a ty se v jádře namapují do souvislého bloku virtuální paměti. FreeBSD nemá a nepotřebuje zóny pro paměť jádra a paměť procesů, jako má Linux, neboť na FreeBSD je možno kteroukoli stránku namapovat do jádra. Výhodou tohoto přístupu je, že odpadá buddy alokátor a alokace velkých bloků je zaručená. Nevýhodou je, že FreeBSD neumí používat velké stránky[2] k mapování paměti jádra (FreeBSD používá velké stránky pouze k mapování kódu jádra, neboť ten je statický). Linux naproti tomu se svým pevným mapováním paměti velké stránky používá.

6.4. Mapování stránek pro DMA

I když máme 64bitový procesor s 64bitovou fyzickou a virtuální adresu, začnou se při překročení hranice 4G objevovat problémy s PCI kartami. Zařízení na sběrnici PCI mohou adresovat paměť pouze 32bitově. Existuje sice i rozšíření PCI pro 64bitové adresování, nicméně to je jednak pomalejší, jednak je nepodporují všechny karty. Na PCI neexistuje žádný DMA řadič podobně, jako třeba na ISA; na PCI si každé zařízení dělá adresaci paměti samo. Tato technika se nazývá Bus-master. Aby se problémům s 4G limitem bylo možno vyhnout, některé 64bitové architektury (např. Sparc64 nebo Alpha) obsahují IOMMU neboli I/O Memory-management unit. Tato jednotka zajišťuje překlad adres podle zadané tabulky mezi 32bitovou PCI sběrnicí a 64bitovou paměťovou sběrnicí. Pokud chce nějaký ovladač provádět DMA přenos mezi pamětí a PCI zařízením, musí paměť namapovat do PCI adresního prostoru. K tomu se na Linuxu používá funkce `dma_addr_t pci_map_single(struct pci_dev *hwdev, void *ptr, size_t size, int direction);`. První parametr je PCI zařízení, druhý je adresa, která se má namapovat, třetí parametr je délka a čtvrtý je typ mapování: `PCI_DMA_TODEVICE`, `PCI_DMA_FROMDEVICE` nebo `PCI_DMA_BIDIRECTIONAL`. Po použití je adresu třeba odmapovat pomocí funkce `void pci_unmap_single(struct pci_dev *hwdev, dma_addr_t dma_addr, size_t size, int direction);`. Existují ještě podobné funkce `pci_map_sg` a `pci_unmap_sg`, které mapují několik paměťových bloků uvedených v seznamu. Na architekturách, které nemají IOMMU, jako například IA32, `pci_map_single` pouze převede virtuální adresu na fyzickou (neboli odečte od ní 3G) a `pci_unmap_single` neudělá nic. Na architekturách s IOMMU `pci_map_single` najde nějakou nepoužitou virtuální adresu v tabulce IOMMU, na ni namapuje požadovanou fyzickou adresu a zpátky předá tuto virtuální adresu. Ovladač předá virtuální adresu PCI zařízení a zařízení na tuto adresu bude dělat DMA. Na architekturách, které nemají automatickou synchronizaci cache procesoru s DMA, tyto funkce navíc synchronizaci cache provádějí.

Na 64bitových architekturách máme díky IOMMU možnost dělat DMA do celé paměti a můžeme se vyhnout dělání paměťových zón. Nicméně programování IOMMU při

[2] Na určitých architekturách (např. IA32 nebo Alpha) je možno pomocí speciálního příznaku u položky adresáře tabulek stránek říct, že tato položka se neodkazuje na tabulku stránek nižší úrovně, ale rovnou na stránku větší velikosti (na IA32 je to 4M nebo 2M). Podobně na Sparc64 se softwarově řízenou TLB je možno do TLB vkládat stránky různé velikosti. Takové větší stránky jsou rychlejší, neboť pak dochází k méně TLB-missům.

každém přenosu (a tedy například při každém packetu ze sítě) zpomaluje. Rozhodně nepovažuji za šťastnou volbu, že IOMMU je povolena i v případech, kdy je v systému méně nebo rovno 4G paměti. Také nemusí být moc příjemné, že se i na architekturách, které IOMMU nemají, musí vyrábět zcela zbytečná struktura `struct scatterlist`, která popisuje mapování nespojitých bloků pro funkce `pci_map_sg` a `pci_unmap_sg`.

FreeBSD neumí používat IOMMU a odtud plyne jeho limit pro maximum 4G paměti. I když jádro FreeBSD by bylo možné snadno přizpůsobit pro libovolné množství paměti, nastanou problémy s I/O zařízeními, neboť ta více než 4G adresovat neumějí. Na architekturách bez IOMMU (jako např. IA32) Linux používá zóny; FreeBSD zóny nemá, stránka může být alokovaná kdekoli, a proto by tam vznikl problém s DMA.

Závěr: Na 32bitových architekturách (např. IA32): Pokud máme méně než 1G paměti, oba systémy to zvládnou bez problémů. Pokud máme 1G až 4G, bude na Linuxu 2.4 zpomalen přístup na disk kvůli zbytečným bounce-bufferům; zatímco na FreeBSD to poběží rychleji. (V případě, že máme méně nebo rovno 2G, tak by asi bylo vhodné modifikovat jádro Linuxu tak, aby mělo 2G virtuální paměti pro procesy a 2G virtuální paměti pro jádro — pak se vyhneme zónám a bounce-bufferům). V případě, že máme více než 4G, je třeba použít Linux, neboť to FreeBSD neumí. Je nutné si uvědomit, že paměť, do které nelze dělat DMA (t.j. na Linuxu nad 1G — při modifikování jádra 2G; FreeBSD dělá DMA do celých 4G), je nepoužitelná jako cache (pokud je už jako cache použita, je hodně pomalá). Je použitelná pouze jako alokovaná paměť procesů, proto by se takové množství paměti mělo instalovat jen tehdy, máme-li program, který ji využije. Je zbytečné dávat takové množství paměti do serveru, neboť tam se většina paměti jako cache používá.

Na 64bitových architekturách: FreeBSD má limit 4G paměti; Linux zvládne libovolné množství paměti bez jakéhokoli zpomalování způsobeného větším množstvím paměti. Drobné zpomalení může přinést programování IOMMU. Paměť je použitelná pro libovolné účely, žádné limity neexistují.

6.5. Alokace struktur v jádře

Stránka je dost velká a v jádře je často potřeba alokovat velmi malé struktury. U alokace struktur se upustilo od alokování z klasické haldy za použití funkcí `malloc` a `free`, protože je to moc pomalé. Byla implementována rychlejší metoda — princip je stejný na obou systémech; na Linuxu se nazývá SLAB CACHE alokátor a na FreeBSD ZONE alokátor (nemá to nic společného se zónami pro alokaci stránek na Linuxu).

Než chce nějaký subsystém v jádře alokovat nějakou strukturu, musí v SLAB alokátoru registrovat typ struktury — v Linuxu se to dělá pomocí funkce `kmem_cache_t *kmem_cache_create(const char *name, size_t size, size_t offset, unsigned long flags, void (* ctor)(void *, kmem_cache_t *, unsigned long), void (* dtor)(void *, kmem_cache_t *, unsigned long))`. První parametr je jméno alokované struktury (použitý při výpisech), druhý parametr je velikost struktury, třetí je offset při barvení (většinou 0 — pak jádro použije automaticky velikost řádky cache), čtvrtý parametr jsou příznaky (většinou debuggovací), pátý parametr je funkce, která inicializuje strukturu, a šestý parametr je funkce, která je volána před uvolněním struktury —

poslední dva parametry mohou být `NULL`, pak se žádné funkce volat nebudou. Funkce vyrobí slab cache, což je v podstatě seznam stránek, a v každé stránce seznam objektů. Ze slab cache je možno strukturu alokovat pomocí funkce `void *kmem_cache_alloc (kmem_cache_t *cachep, int flags)`. Tato funkce alokuje strukturu ze slab cache. `flags` jsou příznaky podobné příznakům `GFP_xxx` — nejpoužívanější je `SLAB_KERNEL` a `SLAB_ATOMIC`. Uvolnit strukturu je možno pomocí funkce `void kmem_cache_free (kmem_cache_t *cachep, void *objp)`. Konstruktor je možno použít k inicializaci některých položek struktury. Je třeba zajistit, aby při uvolnění struktury byly všechny její položky ve zkonstruovaném stavu, neboť konstruktor je volán pouze jednou.

Na rozdíl od funkcí `malloc` a `free` z libc (a jejich typických implementací typu „first-fit“, „next-fit“, „best-fit“ apod.) mají funkce slab alokátoru konstantní složitost — `kmem_cache_alloc` pouze vezme jednu stránku, ze seznamu volných objektů odebere jeden objekt, a ten vrátí. Protože všechny objekty mají stejnou velikost, není třeba hledat blok dostatečné velikosti tak, jak to dělá většina implementací `malloc`. Pokud `kmem_cache_alloc` žádnou stránku s volným objektem nenalezne, alokuje novou stránku a zavolá konstruktor na každý objekt v této stránce. Aby se zjednodušilo hledání stránek, jsou udržovány seznamy plných stránek, částečně plných stránek a prázdných stránek. Z prázdných stránek je alokováno, pokud už nejsou žádné částečně plné stránky, aby se omezila fragmentace paměti. Pokud dochází paměť v systému, jsou prázdné stránky uvolněny. Před tím je ještě na uvolňované objekty zavolán destruktor. Destruktory se příliš nepoužívají. Na SMP jsou použity speciální malé seznamy volných struktur pro každý procesor zvlášť, aby při alokaci nebylo nutno zamykat celou slab cache a nedocházelo k přelévání dat mezi procesorovými cachemi.

Pro lepší využití cache Linux používá barvení. Užitek barvení můžeme vidět na následujícím příkladě: máme velikost řádky cache 32 bytů a velikost struktury 64 bytů. Alokované struktury jsou zřetězeny v seznamu a tento seznam je procházen; při projití se však sáhne pouze na prvních 32 bytů dané struktury. Pokud by struktury byly alokovány bez barvení, tak by tahle operace projití seznamu stresovala sudé adresy v L1 cachi a vůbec by nevyužívala liché adresy. Aby byly všechny adresy cache použity rovnoměrně, objekty v dalších alokovaných stránkách jsou vždy posouvány o velikost řádky cache (nebo o parametr `offset` uvedený v `kmem_cache_create`).

Linux má i funkce `kmalloc` a `kfree`, které jsou ekvivalentní funkcím `malloc` a `free` z libc. Tyto funkce jsou implementovány pomocí slab alokátoru — při startu se předalokují speciální slab cache pro objekty o velikosti mocniny dvojky a `kmalloc` zavolá `kmem_cache_alloc` pro slab objektů příslušné velikosti.

FreeBSD má podobnou implementaci nazvanou ZONE alokátor. FreeBSD neumožňuje použití konstruktorů a destruktorů. Zónu vytvoříme pomocí funkce `vm_zone_t zinit(char *name, int size, int nentries, int flags, int zalloc)`. První parametr je název uvedený ve statistikách, druhý parametr je velikost struktury, třetí je počet předalokovaných položek, čtvrtý je příznak, který určuje, zda se budou objekty alokovat z přerušení (kde není možno čekat; je to ekvivalentní linuxovému `SLAB_ATOMIC`), pátý parametr je počet stránek, co se alokuje, když je zóna prázdná. Většinou jsou třetí a čtvrtý parametr 0 a pátý parametr je 1. Ze zóny se alokuje pomocí funkce `void *zalloc(vm_zone_t z)` a uvolňuje pomocí `void zfree(vm_zone_t z, void *p)`. Zóny na FreeBSD nejsou nikdy zmenšovány. Proto nejsou potřeba seznamy plných/částečně pl-

ných/prázdných stránek. Stačí jen jeden seznam všech volných struktur. FreeBSD nemá barvení; ve zdrojovém kódu je poznámka, že kdysi tam barvení bylo, ale způsobovalo to zpomalení (což je docela dobře možné — při procházení dlouhého seznamu neobarvených struktur se přemaže jen část cache, zatímco při procházení dlouhého seznamu obarvených struktur se přemaže celá).

FreeBSD má rovněž `malloc` a `free` — algoritmus alokace je stejný jako na Linuxu, ale je napsaný zvlášť a nepoužívá zóny.

7. Scheduler

Scheduler je plánovač procesů — rozhoduje, který proces na kterém procesoru poběží. Na přepínání procesů jsou kladeny požadavky, do určité míry protichůdné:

- Nesmí docházet k příliš častému přepínání procesů. Přepnutí procesu je poměrně náročná operace, která nějakou dobu trvá, proto je nežádoucí, aby procesor trávil mnoho času přepínáním procesů místo vykonávání užitečného kódu.
- Nesmí docházet k málo častému přepínání procesů, pak by uživatel získal dojem, že systém reaguje pomalu.
- Pokud nějaký proces čeká na diskové I/O, klávesu na terminálu, data ze sítě nebo jinou událost, musí být probuzen a spuštěn okamžitě, jakmile tato událost nastane. Kdyby například proces četl data z disku a byl spouštěn pozdě, vedlo by to ke značnému poklesu přenosové rychlosti disku.
- Na víceprocesorových systémech je žádoucí, aby proces nestřídal procesory a běžel pokud možno pouze na jednom procesoru. Každý procesor má svoji vlastní cache. Pokud je proces přehozen na jiný procesor, budou jeho data přelévána z cache původního procesoru do cache nového procesoru. Toto přelévání po sběrnici je velmi pomalé.
- Uživatel musí mít možnost nastavit prioritu procesu. U určitých procesů (typicky se jedná o nějaké dlouhodobé výpočty) je požadováno, aby běžely jen, když žádný jiný proces nepožaduje procesor.

Každý proces se může nacházet v několika stavech:

- Proces je zablokován, pokud čeká na nějakou událost. Proces čeká na nějaké čekací frontě, semaforu nebo jiném synchronizačním primitivu. Takový proces není možno spustit. Zablokování máme dvojího druhu: přerušitelné signálem (procesy v tomto stavu jsou označeny písmenem „`S`“ ve výpisech z příkazů `ps` nebo `top`) a nepřerušitelné signálem (jsou označeny písmenem „`D`“). Pokud proces dostane signál a je v přerušitelném čekání, aktuální syscall se zruší, proces se vrátí do userspace a provede se obsluha signálu. Pokud to bylo při registraci signálu požadováno (příznak `SA_RESTART`), pak po skončení obsluhy signálu libc zavolá znovu přerušený syscall. Nepřerušitelné zablokování se používá například při operacích s diskem. Při nich máme zaručeno, že operace skončí v krátkém čase, takže proces nebude příliš dlouho zablokován. Nepřerušitelné zablokování výrazně zjednodušuje implementaci filesystému. Přerušitelné zablokování se používá, pokud čekání může trvat hodně dlouho — při čtení z terminálu a při operacích s pipami a sockety. Scheduler mezi těmito dvěma způsoby zablokování nerozlišuje.
- Proces je připraven, pokud je možno jej spustit, ale neběží (neboť běží jiný proces). Proces se do připraveného stavu dostane buď tak, že běží a je preemptivně přepnut, nebo pokud byl zablokován, probuzen, ale ještě nespuštěn.
- Proces je ve stavu běžící, pokud je jeho kód právě na procesoru vykonáván. Pokud proces běží, může se buď zablokovat, nebo může být preemptivně přepnut, čímž se dostane do stavu připraven.

7.1. Scheduler na Linuxu 2.4 a nižších

Algoritmus plánování procesů na Linuxu je stejný již od první verze 0.01 až po současnou 2.4. Každý proces je popsán velikou strukturou `task_struct`. Položka `state` této struktury určuje stav procesu — může nabývat hodnot `TASK_INTERRUPTIBLE` (přerušitelné zablokování), `TASK_UNINTERRUPTIBLE` (nepřerušitelné zablokování) a `TASK_RUNNING` (proces je připraven nebo běží). Existují dva obousměrně linkované seznamy procesů — seznam všech procesů (položky `next_task` a `prev_task`) a seznam procesů ve stavu `TASK_RUNNING` (položky `next_run` a `prev_run`). Pro účely scheduleru má `task_struct` položky `nice` a `counter`. `nice` je priorita, kterou uživatel nastavil pomocí systémového příkazu `nice`. V příkazu `nice` se nastavuje číslo v rozsahu -20 (největší priorita) až 19 (nejmenší priorita); pro potřeby scheduleru je toto přepočítáno na rozsah 1 (nejmenší priorita) až 40 (největší priorita). `counter` je počet tiků, které procesu ještě zbývají. Při vzniku procesu se `nice` zkopíruje do `counter`.

Jádro scheduleru se nachází ve funkci `void schedule(void)`. Tato funkce odstaví aktuální proces ze seznamu připravených procesů, pokud jeho stav není `TASK_RUNNING`. Pak vybere ze seznamu připravených procesů proces s největší hodnotou `counter` a ten spustí. Při běhu procesu se `counter` zmenšuje o jedničku při každém tiku. Proces běží, dokud `counter` nedosáhne nuly nebo dokud se sám nezablokuje. Pak se opět zavolá `schedule` a ta vybere další proces. Pokud funkce schedule zjistí, že všechny připravené procesy mají `counter` roven nule, provede pro všechny (připravené i zablokované) procesy operaci `counter = counter / 2 + nice`. Pro připravené procesy tato operace pouze přiřadí `nice` do `counter`, pro zablokované procesy způsobí, že `counter` bude pomalu růst až k hodnotě `2 * nice`. Tím je zajištěno, že proces, který je delší dobu zablokovaný, bude mít vyšší prioritu než všechny běžící procesy a po probuzení bude spuštěn okamžitě. Na starších jádrech byla hodnota `nice` používána rovnou (což znamenalo, že při tiku 100Hz dostal proces s implicitní prioritou časové kvantum 200ms), v novějších 2.4 je lineárně přepočítávána tak, aby proces s implicitní prioritou měl přibližně 50ms časové kvantum. Ač je tento algoritmus velmi jednoduchý, splňuje většinu požadavků na scheduler kladených: pokud běží několik procesů, střídají se a dostávají časová kvanta úměrná jejich prioritě. Pokud je nějaký proces delší dobu zablokován, naroste mu `counter` na větší hodnotu než všem ostatním procesům, je po probuzení spuštěn okamžitě a má až dvakrát větší časové kvantum. Pokud je proces zablokován jen na velmi krátkou dobu, tak mu `counter` příliš nenaroste.

Horší je to už s víceprocesorovými systémy. Byla snaha zabránit přehazování procesorů mezi procesory tak, že při výběru procesu s největším `counter` je k této hodnotě přičteno 5, pokud proces běžel naposledy na procesoru, na kterém se schedulování provádí. Rovněž je přičteno 1, pokud proces má stejnou virtuální paměť jako proces, který na procesoru právě běžel — neboli pokud se jedná o thready. Přepínání mezi thready je méně náročné než přepínání mezi procesy. Toto nám zajišťuje, že pokud budeme mít dvouprocesorový systém a na něm nám poběží sudé množství procesů, bude mít každý proces svůj procesor a procesy nebudou mezi procesory přehazovány. Nicméně pokud nám na dvouprocesorovém systému poběží tři procesy, tak k cyklickému přehazování jednoho nebo všech procesů mezi procesory dojde. Optimální strategie v takovém případě by byla nechat na jednom procesoru běžet dva procesy, na druhém procesoru jeden proces

a přehození procesu provádět pouze za dlouhou dobu, aby všechny procesy dostaly stejné procento výpočetní síly.

Další nevýhodou tohoto scheduleru je jeho veliká časová složitost. Každé přepnutí procesu má složitost lineární vzhledem k počtu připravených procesů. To není ještě tak zlé — připravených procesů je v systému jen pár; systém s více jak dvaceti připravenými procesy je stejně nepoužitelný (neboť tam každý proces dostane pouze dvacetinu výpočetního času), takže uživatelé tak velké zátěže na systém nedávají. Nicméně pokud všem dojde `counter` na 0, musí se projít všechny procesy v systému, a to je značně náročná operace, neboť procesů může být až několik stovek. Složitost jednoho přepnutí procesu je tedy O(počet připravených procesů + počet všech procesů / počet připravených procesů).

Linux má realtimové procesy. Pokud v systému existuje nějaký připravený realtimový proces, je spuštěn přednostně před všemi ostatními procesy. K nastavení realtimové priority procesu je potřeba právo superuživatele, neboť tyto procesy jsou poměrně nebezpečné; pokud se v realtimovém procesu vyskytne nekonečná smyčka, celý systém zatuhne. Realtimové procesy mají uživatelsky nastavené priority, spuštěn je vždy realtimový proces s největší prioritou, pokud je více realtimových procesů se stejnou prioritou, tak se mezi nimi cyklicky přepíná. Realtimové procesy nejsou realtimové v pravém slova smyslu — je sice garantováno, že realtimový proces bude spuštěn před všemi ostatními procesy, nicméně doba spuštění garantována není. Pokud například nějaký nerealtimový proces stráví příliš mnoho času v jádře, aniž by se zablokoval nebo zavolal funkci podmíněného přepnutí `cond_resched`, tak realtimový proces být spuštěn nemůže. K vylepšení doby odezvy je třeba použít low-latency patch nebo preemptivní přepínání procesů uvnitř jádra.

7.2. Scheduler na Linuxu 2.5

V experimentálním Linuxu 2.5 byl scheduler zcela přepsán, aby měl složitost O(1) a aby lépe fungoval na SMP. Scheduler pracuje zvlášť na každém procesoru a má frontu zvlášť pro každý procesor. Každý proces má proměnnou `sleep_avg`, která se zvětšuje o 1 každý tik, když proces spí, a zmenšuje se o 1 každý tik, když proces běží. Má určitou maximální velikost, kterou nesmí překročit. Efektivní priorita procesu je určena jako statická priorita nastavená uživatelem +/− bonus, kde bonus je `sleep_avg` lineárně zobrazený do intervalu $-5 - +5$. Připravené a běžící procesy jsou uloženy ve frontách příslušejících jejich efektivním prioritám. K frontám existuje bitová maska neprázdných front, aby se mohla snadno najít neprázdná fronta s nejvyšší prioritou. Pro každou prioritu existují dvě fronty: aktivní a záložní. Každý proces dostane `time_slice`, což je uživatelem nastavená priorita lineárně zobrazená do intervalu 10ms – 300ms. Systém vybere z aktivních front proces s nejvyšší efektivní prioritou a pustí ho. Proces běží a ubírá svůj `time_slice`. Až `time_slice` dojde, proces je přeřazen z aktivní fronty do záložní fronty příslušející jeho prioritě. Pak je vybrán další proces s nejvyšší prioritou z aktivních front. Když už v aktivních frontách žádné procesy nejsou, tak se záložní a aktivní fronty vymění a provede se znovu výběr procesu.

Pokud je úloha interaktivní (což je rozhodnuto lineárně na základě `sleep_avg` a uživatelské priority), je při ztrátě `time_slice` opět vložena do aktivní fronty. V takovém

případě by mohlo docházet k hladovění procesů v záložní frontě — pokud k tomu dojde (což se pozná tak, že proces v záložní frontě s nejvyšší prioritou neběžel po určitou dobu), tak se interaktivní procesy budou dávat do záložní fronty jako obyčejné procesy.

Aby scheduler dobře fungoval na SMP, celý tenhle mechanismus funguje zvlášť na každém procesoru. Jednou za 250ms, nebo pokud je nějaký procesor volný, je prováděn load-ballancing, při kterém se systém snaží vyrovnat velikost front. Procesor, na kterém se load-ballancing provádí, najde nejvytíženější procesor. Pokud je tento procesor vytížen více jak na 4/3 zátěže procesoru provádějícího load-ballancing, je mu nějaký proces odebrán.

Tento scheduler je určitě výrazně lepší než ve starších verzích Linuxu: všechny operace zde mají složitost O(1); procházení seznamu všech procesů nebo seznamu všech připravených procesů se zde nedělá nikde. Rovněž je zajištěna dobrá funkce na SMP, neboť procesy nejsou svévolně přehazovány mezi procesory. Procesy jsou přehazovány jen při operaci load-ballancingu.

7.3. Scheduler na FreeBSD 4 a nižších

Scheduler na FreeBSD funguje následovně — máme 32 front příslušejících prioritám. Scheduler vybírá procesy z fronty s nejvyšší prioritou a mezi nimi cyklicky přepíná. Každý proces běží 100ms. Každý proces má proměnnou `p_estcpu`. Tato proměnná roste lineárně, když proces běží, a klesá exponenciálně, když proces neběží. Rychlost klesání je ovlivněna systémovou zátěží (load average, vypsáno příkazem `uptime`). Za 5*`load_average` sekund `p_estcpu` klesne na 10%. Skutečná priorita je určená lineárně z uživatelské priority a `p_estcpu`. Scheduler jednou za sekundu prochází všechny procesy a snižuje `p_estcpu`, čímž zvyšuje prioritu neběžícím procesům. Tím je zajištěno, že procesy nehladoví. Pokud proces čeká na nějakou událost pomocí funkce `tsleep` nebo `asleep`, může v parametru funkce určit, s jakou prioritou bude probuzen. Scheduler přepíná procesy se složitostí O(1), ovšem jednou za sekundu prochází všechny procesy. Nicméně funkce `wake_up` prochází všechny procesy při každé události probuzení — optimalizování by se mělo nejdříve provést tam, neboť `wake_up` se volá mnohem častěji než scheduler.

Scheduler na FreeBSD 4 na SMP obsahuje naprosto neuvěřitelný bug (o kterém jsem se již zmiňoval v kapitole o SMP), který způsobí, že pokud jsou všechny procesory obsazeny, mají interaktivní procesy až několikasekundovou odezvu.

7.4. Kernel schedulable entities na FreeBSD 5

Na FreeBSD 5 byl scheduler zcela přepsán, aby lépe umožňoval použití threadů. Nebyl změněn jen algoritmus, ale byl změněn i význam datových struktur.

Existují dvě základní metody, jak implementovat thready:

- **Kernel thready** — thready přepínané v jádře. Jádro se ke threadům chová stejně jako k samostatným procesům, které mají společnou virtuální paměť, a přepíná je. Takhle jsou thready implementovány na Linuxu a na starších FreeBSD.

- **User thready** — jádro vidí jeden proces, přepínání mezi thready provádí samotná threadová knihovna běžící v userspace. Takhle fungovala například dnes již zapomenutá knihovna NSPR na Linuxu, kterou používal Netscape.

User thready mají nevýhodu, že pokud se jeden thread dostane do nepřerušitelného zablokování, žádné další thready nemohou běžet, protože nemůže být doručen časovací signál knihovně, která thready přepíná. Implementace přerušení přerušitelného čekání je značně problematická. Navíc user thready nemohou efektivně využívat víceprocesorový stroj, neboť pro jádro se celý mnohothreadový program jeví jako jeden proces, který je spuštěn na jednom procesoru.

Naproti tomu kernel thready mají nevýhodu, že pokud thready čekají na sebe navzájem, musí se přitom provádět volání jádra, což se při user threadech nemusí. User thready také narozdíl od kernel threadů nekonzumují žádné zdroje jádra.

Na FreeBSD byla snaha napsat hybridní thready, které by měly výhody jak kernel, tak user threadů. Idea je taková, že se thready budou chovat jako thready userspacové, ale pokud se nějaký thread zablokuje v jádře, bude vyrobena nová struktura popisující kontext jádra a ostatní thready procesu poběží s touhle novou strukturou, zatímco stará struktura zůstane zablokovaná v jádře. Proto byl zcela změněn význam některých struktur jádra a přibyly struktury nové. `struct proc` popisuje proces stejně jako v předchozích verzích. Nicméně tato struktura již neobsahuje informace pro scheduler a neobsahuje kontext jádra (t.j. zásobník jádra a příznaky související se zablokováním a odblokováním). Kontext jádra byl přesunut do struktury `struct thread`. Proces bude mít několik struktur `struct kse` (kernel schedulable entity) — pro každý procesor jednu. Ty se budou z hlediska schedulování chovat jako samostatné procesy. Každá tato struktura má jednu `struct thread` obsahující kontext jádra. Pokud se tento kontext zablokuje někde v jádře, bude alokována nová `struct thread` a s ní daná KSE dále poběží. KSE jsou sdruženy v `struct ksegrp`, která obsahuje obecné informace pro scheduler, jako například `p_estcpu`. Uvnitř jednotlivých KSE pak poběží samotné userspacové thready, přepínané schedulerem v userspace. To zajistí výhody rychlého přepínání mezi user thready bez vzniku problémů s blokováním threadů v jádře a s víceprocesorovými stroji.

Současný stav je asi takový, že byly jednotlivé položky ze `struct proc` rozděleny do `struct proc`, `struct kse`, `struct thread` a `struct ksegrp` a většina funkcí scheduleru byla přepsána, aby místo `struct proc` používala `struct thread`. Jinak v aktuální verzi stále platí mapování 1:1:1:1 mezi těmito strukturami — t.j. to, co jsem popsal v předchozím odstavci, vůbec nefunguje a je jen vizí vývojářů, že to takhle jednou fungovat bude. Userspacová knihovna na thready taktéž napsána není.

Pokud bude tenhle model dopsán, bude to znamenat, že uživatel bude mít možnost pouštět velké množství threadů, aniž by tím zatěžoval paměť jádra, a že když thready na sebe budou navzájem čekat, bude jejich probouzení výrazně rychlejší. Pokud bude pouštěno velké množství threadů, které budou zablokované v jádře (například u serveru thready čekající na socketech posílající nebo čtoucí data od klientů), tak tenhle threadový model oproti ryzím kernel threadům žádné zrychlení nepřinese — thready budou stejně potřebovat kontext jádra (a s ním `struct thread`), a protože na sebe thready nebudou čekat, tak se výhody threadů v userspace neprojeví.

7.5. Měření rychlosti správy procesů

Byly provedeny benchmarky, měřící rychlost systémových volání `fork` a `exec`. Na Linuxu 2.4.20 `fork` trvá 1.5274ms, na FreeBSD 4.7 1.4730ms. Dvojice syscallů `fork`+`exec` (jako parametr syscallu `exec` byl předán prázdný program, který ihned skončí) trvá na Linuxu 2.4.20 6.9778ms a na FreeBSD 4.7 6.2069ms.

Bylo provedeno měření rychlosti přepnutí procesu. Měření bylo provedeno tak, že byly spuštěny dva procesy, které periodicky volaly instrukci `rdtsc`. Pokud byl rozdíl dvou po sobě navrácených časů moc veliký (tzn. došlo k přepnutí), proces vypsal svoje PID a předchozí i aktuální hodnotu `rdtsc`. Z výpisu je možno snadno poznat, jak dlouho přepnutí procesu trvá. Na Linuxu 2.2.20 trvá přepnutí procesu v průměru 0.028ms, nejvyšší hodnota byla 0.032ms a nejnižší 0.022ms. Na FreeBSD 4.7 trvá přepnutí procesu v průměru 0.041ms, nejvyšší hodnota je 0.063ms a nejnižší 0.036ms.

8. VFS — rozhraní pro přístup k filesystému

Aby jádro mohlo pracovat s více druhy filesystémů, bylo uděláno rozhraní mezi jádrem a ovladačem filesystému nazvané VFS (virtual filesystem). VFS je soubor funkcí, které může ovladač filesystému volat, a soubor funkcí, které musí jádru poskytnout. Nejstarší část VFS je bufferová cache. Bufferová cache existuje již v původním Unixu z Bellových laboratoří, z dob, kdy ještě žádné VFS neexistovalo a jádro mělo v sobě napevno „zadrátovaný“ jeden filesystém[1].

8.1. Bufferová cache

Každý buffer má hlavu (struktura `struct buffer_head` na Linuxu a `struct buf` na FreeBSD) a datovou oblast. Hlava obsahuje různé příznaky a informace o bufferu (např. blokové zařízení, ke kterému buffer náleží, číslo bloku, zda jsou data platná, zda je potřeba buffer uložit na disk, zda je právě prováděno čtení nebo zápis dat a podobně). Hlava také obsahuje pointer na datovou oblast. V datové oblasti se nacházejí data načtená z disku pro příslušný blok. Existuje hashová tabulka všech bufferových hlav, ve které je možno podle blokového zařízení a čísla bloku vyhledat buffer. Základní operace na bufferové cachi jsou

- `struct buffer_head *bread(blokové zařízení, číslo bloku, velikost bloku)` — tato funkce vyhledá v hashové tabulce buffer. Pokud buffer najde, zkontroluje, zda je na něm právě prováděna operace čtení. Pokud ano, počká na příslušné frontě, až operace skončí. Pokud operace čtení neprobíhá, zkontroluje, zda jsou data platná. Pokud ano, vrátí buffer, pokud ne, spustí tato funkce sama operaci čtení a počká. Pokud buffer není nalezen v hashové tabulce, je vytvořen, vložen do tabulky, je na něm zahájena operace čtení a funkce počká, než tato operace skončí. Po návratu této funkce je buffer v zamčeném stavu — t.j. nemůže být uvolněn z paměti.
- `void brelse(struct buffer_head *)` — odemkne buffer dříve zamčený funkcí `bread`. Jádro může odemčený buffer kdykoli uvolnit (dělá to zpravidla, pokud dochází paměť). Proto se strukturou `buffer_head`, která ukazuje na odemčený buffer, již není možno dále pracovat.
- `void mark_buffer_dirty(struct buffer_head *)` na Linuxu a ekvivalentní `void bwrite(struct buf *)` na FreeBSD — označí buffer jako modifikovaný. Pokud kód filesystému zavolá `bread`, může pak obsah datové oblasti bufferu modifikovat a po provedení tohoto modifikování musí zavolat zmíněnou funkci. Tím jádru dává najevo, že je potřeba, aby buffer byl uložen zpátky na disk. Buffer není zpravidla uložen hned (pokud není zapnutý synchronní zápis na filesystému), ale je uložen až někdy později, aby filesystém zbytečně nemusel čekat na dokončení operace zápisu.
- `void bforget(struct buffer_head *)` — funguje podobně jako `brelse`, až na to, že buffer okamžitě uvolní z cache, a pokud je modifikovaný, nezapisuje ho na disk. Používá se poté, co filesystém odalokoval nějakou strukturu na disku po provedení

[1] Původní Linux měl také zadrátován pouze jeden filesystém (minixfs). VFS a podpora dvou filesystémů (minixfs a extfs) se objevily až v jádře 0.96c.

`unlink`, `truncate` nebo `rmdir` — v takovém případě již nemá smysl uvolněná data dále držet v cachi nebo je zapisovat na disk, protože se na ně filesystém již neodkazuje.

Bufferová cache umožňuje ovladači filesystému velmi efektivně pracovat s daty na disku. Pokud se buffer nachází v cachi, je operace `bread` velmi rychlá — v podstatě se provede pouze vyhledání v hashové tabulce a zvýšení počítadla zámků. Nedochází přitom ke kopírování dat. Taktéž další operace — `brelse` a `mark_buffer_dirty` — jsou velmi rychlé.

Linux umí na jednom zařízení pracovat pouze s buffery stejné velikosti (tuto velikost specifikuje ovladač filesystému při mountování). FreeBSD umožňuje různé velikosti bufferů — to ovšem činí kód funkcí pro operace s buffery značně komplikovaný. Linux má maximální velikost bufferu jedna stránka. FreeBSD má maximální velikost bufferu 64k. Buffer zde může zabírat několik stránek, které jsou namapovány do souvislé oblasti ve virtuální paměti jádra. Na FreeBSD je při startu systému alokován pevný počet bufferových hlav a hlavy již nemohou přibývat (ale paměť alokovaná pro datovou oblast bufferů se může zvětšovat i zmenšovat). Linux umí bufferové hlavy alokovat a uvolňovat za běhu podle potřeby.

8.2. Inodová cache

Bufferová cache by sama o sobě stačila k omezení přístupu na disk. Na původním Unixu to byla jediná cache. Postupem času se však začaly objevovat další cache (inode, dentry a page cache), které umožňují rychlejší přístup k vyšším strukturám filesystému. Na Linuxu je cache nazývána inodová; na FreeBSD vnodová. Pod pojmem „inode“ se na Linuxu rozumí jak inoda uložená na disku, tak inoda uložená v paměti v cachi. Na FreeBSD se pojmem „inode“ označuje pouze inoda na disku; inoda v cachi se nazývá „vnode“.

Inoda je objekt, který přísluší každému souboru a adresáři na filesystému. Inoda obsahuje různé informace o souboru — velikost, práva přístupu, časy vytvoření/modifikace/přístupu, informace o umístění datových bloků na disku a podobně. Inoda neobsahuje jméno souboru ani ukazatel na nadřazený adresář (aby bylo možno dělat hard-linky).

Inodová cache je hashová tabulka, ve které je možno podle dvojice (blokové zařízení, číslo inody) inodu vyhledat. Inoda má jednotný formát, ve kterém je v paměti v této cachi (na Linuxu `struct inode`, na FreeBSD `struct vnode`). Ovladač filesystému musí poskytnout dvě funkce — jednu, která přečte inodu z disku a překonvertuje ji do formátu v paměti, a druhou, která zapíše inodu nacházející se v paměti na disk. Tyto funkce používají bufferovou cache pro operace s diskem. Inoda v paměti má příznaky, zda je modifikovaná (a je tedy třeba ji zapsat), nebo zda je právě načítána z disku, což znamená, že položky jsou neplatné a nesmí se používat. Správu těchto příznaků, jakož i volání oněch funkcí ovladače filesystému, zajišťuje jádro. Na Linuxu dělá hashování inod samotné jádro; na FreeBSD dělá hashování vnod ovladač filesystému. Linux umí paměť obsaženou pro inody zmenšovat a zvětšovat podle potřeby; na FreeBSD dochází pouze ke zvětšování vnodové paměti až do uživatelem nastaveného limitu. Paměť vyhrazená pro vnody na FreeBSD není nikdy uvolňována[2].

[2] V některých částech jádra FreeBSD se s vnodami pracuje takovým způsobem, že si

8.3. Cache pro vyhledávání v adresářích

Původně cache pro vyhledávání neexistovala. Ovladač filesystému poskytoval funkci `lookup`, která dostala jako parametr inodu(resp. vnodu) adresáře a jméno souboru a vracela inodu souboru s daným jménem v daném adresáři. Inodová cache zabránila čtení inod z disku, ale bylo třeba také zabránit pomalému vyhledání v adresáři. Na vyhledávání souboru v adresáři se používá bufferová cache, která sama o sobě zabrání opakovanému čtení z disku. Nicméně vzhledem k tomu, že adresář je sekvence dvojic (jméno souboru, inoda), je lineární prohledávání adresáře v bufferech velmi pomalé.

Na Linuxu byla původně hashová tabulka pro vyhledávání jmen k adresářům v ovladači filesystému — což s sebou přinášelo tu nevýhodu, že bylo třeba psát cache pro každý filesystém znovu. Na Linuxu 2.2 byla zavedena nová dentry cache, která funguje obecně pro všechny filesystémy. Každý filesystém obsahuje strom struktur `struct dentry`. Každá dentry obsahuje jméno souboru, rodičovskou dentry a ukazuje na inodu, které náleží (ale ne každá inoda musí mít dentry — dentry je možno uvolnit a inodu zachovat v paměti). Adresářová dentry má hashovou tabulku obsahující dentry příslušející položkám adresáře. Při vyhledávání souboru se nevolají žádné funkce filesystému, ale prochází se strom dentry. Když není dentry daného jména nalezena, vytvoří se prázdná dentry a zavolá se funkce filesystému `lookup`, která soubor nalezne a dentry vyplní. Při dalším hledání téhož souboru se již použije vytvořená dentry. Pokud funkce `lookup` jméno nenalezne, vytvoří tzv. negativní dentry. Negativní dentry informuje o tom, že soubor daného jména se v adresáři nevyskytuje. Když je při prohledávání stromu nalezena negativní dentry, tak se okamžitě vrátí chyba. Pokud se tedy program bude snažit opakovaně otevřít neexistující soubor, bude tato operace otevření velmi rychlá (v podstatě jen vyhledání v hashové tabulce) a nebude se při tom muset vyhledávat v adresáři. Při nedostatku paměti se uvolňuje jak dentry cache, tak inode cache. Je třeba zajistit, že nebude uvolněna inoda, na kterou ukazuje dentry, a nebude uvolněna dentry, která není listem stromu (t.j. má v hashové tabulce nějaké dentry). Se zavedením dentry cache definitivně padla možnost dělat hard-linky na adresáře, neboť kód jádra předpokládá, že dentry tvoří strom.

FreeBSD má podobnou cache jako Linux (i když byla udělána později než na Linuxu). Cache na FreeBSD nemá nutně podobu stromu — cache je udělána takovým způsobem, že ke každé adresářové vnodě je možno v cachi nalézt seznam souborů a podadresářů. Cache může obsahovat i negativní položky pro nenalezené soubory. Položky cache jsou alokovány pomocí funkce `malloc`, takže se po uvolnění vracejí do malloc poolu. Dříve FreeBSD sjednocenou vyhledávací cache nemělo, a tak byla cache napsána přímo do filesystému — tato cache tam pořád zůstala, takže v současné době jsou ve FreeBSD dvě cache obsahující tatáž data. Cache ve filesystému je možno vypnout (odstranit `UFS_DIRHASH` v konfiguračním souboru kompilace jádra), neboť je zbytečná.

---

kód jádra zapamatuje pointer na vnodu, blokové zařízení a číslo vnody — a za dlouhou dobu (aniž by mezitím držel nějaký zámek na vnodě) se na pointer podívá, zjistí, zda je vnoda platná, porovná zařízení a číslo, a pokud souhlasí, začne s vnodou pracovat. Kdyby se paměť pro vnody uvolňovala, přestane tenhle mechanismus přestane fungovat a náhodné bloky paměti budou považovány za vnody.

8.4. Stránková cache

S buffery se pracuje tak, že pokud kód chce přečíst data z disku, zavolá `bread`, jádro data někam načte a vrátí na ně pointer (nebo vrátí pointer na už načtená data, pokud jsou v bufferové cachi). Tento způsob práce minimalizuje kopírování dat, pokud jsou data už nacachovaná. Nevýhoda spočívá v tom, že sama bufferová cache si určuje, kam data umístí. Pokud máme soubor namapovaný pomocí `mmap` (nebo spuštěný program — spouštění programů se taktéž dělá pomocí `mmap`), tak je třeba, aby určité bloky byly nataženy v jedné stránce, aby se snadno daly namapovat do uživatelského adresního prostoru. Proto byla zavedena další cache — page cache. Page cache se používá pro stránky obsahující data souborů. Page cache je hashová tabulka, ve které je možno pomocí dvojice (inoda resp. vnoda, offset stránky) vyhledat stránku náležící danému souboru.

Původní Unix neměl virtuální paměť, a proto neměl ani page cache. Měl pouze bufferovou cache. Když byla virtuální paměť do BSD Unixu připsána, byla page cache zcela oddělená od bufferové cache. Při natahování mmapovaných stránek se data nejdříve načetla do bufferové cache a odtud se zkopírovala do stránek. OpenBSD, NetBSD, OS/2 a mnohé komerční Unixy to tak mají dodnes. Toto kopírování stránek je zcela zbytečné a způsobuje zpomalení. Aby se kopírování dat zabránilo, tak se page cache a buffer cache prolíná. Pokud se mají načíst nějaká data souboru do stránky, alokuje se stránka, k této stránce se alokuje příslušný počet bufferových hlav (na Linuxu `struct buffer_head`, na FreeBSD `struct buf`) a datová oblast těchto bufferů se nechá ukazovat na části stránky. Poté se buffery předají ovladači pro blokové zařízení, aby načetl data. Tento přístup zabraňuje kopírování dat, ale vede k další nepříjemnosti — spoustě práce s bufferovými hlavami. V podstatě to vypadá tak, že se při čtení několika stránek alokuje bufferová hlava pro každý blok, pak se pro každou tuto hlavy zavolá funkce čtení bufferu, tato funkce začne vyrábět požadavky na blokové zařízení a zjistí, že buffery ukazují na souvislou část disku, a tak je spojí do jednoho velkého požadavku. Rozdělování na jednotlivé bufferové hlavy a jejich spojování je zcela zbytečné. V experimentálním Linuxu 2.5 byl tento problém vyřešen — byl zaveden zcela nový model posílání požadavků na bloková zařízení — pomocí `struct bio` je možno poslat jeden požadavek na několik (až 16) v paměti nesouvislých stránek. V souboru `mpage.c` je vidět alokace stránek a jejich čtení nebo zápis pomocí `struct bio`. Pokud jsou data souboru na disku nesouvislá, tak se tento model nepoužívá a používají se staré bufferové hlavy.

8.5. Direct IO

Cache byla vytvořena proto, aby urychlovala přístup k datům. Existují však situace, kdy je cache nežádoucí a naopak přístup zpomaluje. Cache je nežádoucí ve dvou případech: u aplikací pracujících s proudem velkého množství dat (například digitální video), kde se všechna data do paměti stejně nevejdou a při sekvenčním přístupu se z cache žádná data číst nebudou. Druhým případem, kde je cachování nežádoucí, jsou databázové aplikace — databázové servery dělají vlastní cachování, které je efektivnější než systémová cache, neboť databázový server zná strukturu dat a umí lépe předpovídat,

ke kterým datům se bude přistupovat. Data není třeba cachovat dvakrát — jednou na úrovni databázového serveru a podruhé na úrovni operačního systému.

Direct IO je způsob, jak dělat čtení nebo zápis dat bez použití cache. Cílem direct IO je nepoužívat paměť pro cachování dat, která stejně nemá smysl cachovat, a také zabránit kopírování dat ze systémové cache do uživatelského adresního prostoru procesu. Při direct IO jsou data přenášena přímo mezi diskem a adresním prostorem procesu, který zavolal `read` nebo `write`. Pokud uživatel nastaví příznak `O_DIRECT` v syscallu `open`, budou všechny čtení a zápisy na daném souboru dělány pomocí direct IO.

Poslední verze Linuxu 2.4 podporují direct IO. Direct IO je implementováno pomocí tzv. kiobufů. `struct kiobuf` je struktura popisující stránky zamčené v adresním prostoru procesu. Stránky je možno zamknout pomocí funkce `int map_user_kiobuf(int rw, struct kiobuf *buf, unsigned long va, size_t len)`. První parametr určuje, zda se jedná o čtení, nebo zápis, druhý parametr je ukazatel na kiobuf, třetí parametr je adresa v uživatelském adresním prostoru a čtvrtý parametr je délka. Funkce zamkne v paměti stránky v daném rozsahu (případně je naswapuje, pokud jsou odswapované). Odemčení stránek dělá funkce `void unmap_kiobuf(struct kiobuf *buf)`. Direct IO na Linuxu tenhle mechanismus používá k zamčení stránek a provedení přímého čtení nebo zápisu do adresního prostoru procesu. Andrea Arcangeli změřil, že direct IO způsobí, že čtení dat spotřebuje desetkrát méně CPU času než čtení bez direct IO. Čtení bez direct IO je tak pomalé proto, že se data nejprve načtou do systémové cache a teprve poté se kopírují do adresního prostoru procesu.

FreeBSD rozumí příznaku `O_DIRECT`, nicméně pravé direct IO nemá. Na FreeBSD jsou data při použití `O_DIRECT` stále přenášena z disku do cache a z cache do adresního prostoru procesu. `O_DIRECT` pouze způsobí, že stránky náležící danému souboru budou brzy uvolněny z cache.

9. Filesystémy

9.1. Klasický unixový filesystém

Návrh linuxového filesystému Ext2 i filesystému FreeBSD UFS pochází z klasického unixového filesystému. Filesystém začíná superblokem, který obsahuje obecné informace — velikost filesystému, množství inod, inodu kořenového adresáře a podobné. Každému souboru a adresáři na klasickém unixovém filesystému odpovídá inoda. Inody jsou předalokovány ve speciální části filesystému, místo pro inody je určeno při vytváření filesystému a je dále neměnné. Velikost místa pro inody určuje maximální počet souborů a adresářů na filesystému. Místo, které se nepoužívá pro inody, se používá pro bloky. Velikost bloku je určena při vytváření filesystému. Inoda obsahuje všemožné informace o souboru nebo adresáři — délku, časy modifikace/vytvoření/čtení, číslo uživatele a skupiny, práva přístupu a podobně. Inoda neobsahuje jméno souboru.

Inoda obsahuje 12 ukazatelů na prvních 12 bloků souboru. Pokud je soubor delší, ukazuje třináctý ukazatel na tzv. indirect block první úrovně — tento blok obsahuje ukazatele na další bloky inody. Počet ukazatelů v indirect bloku závisí na velikosti bloku. Pokud je soubor ještě delší, ukazuje čtrnáctý ukazatel v inodě na indirect blok druhé úrovně. Tento blok obsahuje ukazatele na další bloky, které teprve obsahují ukazatele na bloky souboru. Pokud je soubor tak dlouhý, že se na všechny bloky nepodaří ukázat ani pomocí indirect bloku druhé úrovně, použije se ukazatel na indirect blok třetí úrovně — pod ním jsou tři vrstvy bloků s ukazateli a pouze bloky poslední vrstvy ukazují na bloky souboru. Unixový filesystém umožňuje dělat v souborech díry. Pokud například uživatelský program zavolá syscally `creat`, `lseek(10000)`, `write`, tak bude alokován pouze blok příslušející pozici 10000. Předchozí ukazatele budou mít hodnotu nula. Při jejich čtení jádro samo vrátí stránky plné nul, aniž by četlo cokoli z disku.

Adresáře jsou spravovány stejně jako soubory. Adresář je soubor obsahující sekvenci dvojic (jméno souboru, číslo inody). Je možno, aby na jednu inodu souboru ukazovalo několik položek adresářů — tato situace se nazývá hard-link. Není možno dělat hard-linky na adresáře.

K popisu volných bloků a volných inod se používají bitmapy. Každý bit v bitmapě odpovídá jednomu bloku (resp. jedné inodě) na filesystému. Z hlediska teoretické informatiky je sice bitmapa značně nevýhodná struktura pro popis volných bloků, ale v praxi se ukázala lepší než zřetězené seznamy volných bloků. Seznamy volných bloků mohou značně nabývat na velikosti (to lze sice řešit tak, že seznam uložíme do samotných volných bloků, ale to zase značně zpomaluje přístup a znemožňuje spojování volných bloků).

9.2. Rozšíření klasického unixového filesystému — Ext2 a UFS

Nejpoužívanějším filesystémem na Linuxu je Ext2 (čteno „second extended filesystem“). Struktura Ext2 je založena na klasickém unixovém filesystému. Filesystém byl rozdělen na tzv. skupiny (groups). Skupiny mají velikost určenou při vytváření filesystému, implicitně je to 8M. Na začátku každé skupiny se nacházejí inody, bitmapy pro

inody a pro data téhle skupiny a kopie superbloku (použije se v případě, že byl hlavní superblok zničen). Rozdělení na skupiny omezuje přesouvání diskové hlavy a způsobuje rychlejší přístup. Pokud například z adresáře přistupujeme na inodu, je velká pravděpodobnost, že tato inoda bude ve stejné skupině, a že bude tedy rychle načtena; podobně když z inody přistupujeme na data, je pravděpodobné, že budou v téže skupině.

FreeBSD používá filesystém UFS (a je to jediný použitelný diskový filesystém na FreeBSD – ostatní filesystémy, které FreeBSD umí, jsou buď velmi jednoduché (FAT), plné chyb (Ext2 pro FreeBSD), nebo read-only (HPFS, NTFS, UDF). UFS má rovněž disk rozdělen na skupiny podobně jako na Linuxu.

Dalším rozšířením UFS jsou fragmenty. Fragmenty lépe řeší problém velikosti bloku — pokud je blok moc malý, je filesystém pomalý, neboť bude hodně práce s bufferovou cachí a s vyhledáváním v indirect blocích. Pokud je blok moc velký, budou malé soubory zabírat zbytečně mnoho místa. UFS řeší tento problém tak, že používá velkou velikost bloku, některé bloky rozděluje na menší fragmenty a do těch ukládá malé soubory. Velikost fragmentu je nejméně osmina velikosti bloku. Velikost bloku může být na UFS větší než velikost stránky (neboť bufferová cache na FreeBSD zvládá buffery větší než stránka; na Ext2 je maximální velikost bloku rovna velikosti stránky procesoru, na kterém Linux běží). V bitmapách jeden bit odpovídá jednomu fragmentu. Pro každou skupinu navíc existuje pole o velikosti 8, které říká, kolik volných fragmentů dané velikosti se ve skupině nachází. Pokud je třeba alokovat fragment dané velikosti, je vybrána skupina, ve které se fragment příslušné velikosti nalézá, a pak je fragment nalezen v bitmapě[1]. Při rozšiřování souboru alokovaného ve fragmentu FreeBSD buď alokuje větší fragment příslušné velikosti (optimalizace na místo na disku), nebo alokuje rovnou celý blok (optimalizace na rychlost — pokud se alokuje celý blok, tak ho již dále není třeba realokovat)[2]. Filesystém sám mezi těmito dvěma algoritmy přepíná — pokud se filesystém začne zaplňovat, přepne na alokaci fragmentu přesné velikosti a do logu napíše `optimization changed from TIME to SPACE`. Až je na filesystému více volného místa, tak přepne zpátky.

9.3. Algoritmus alokace místa na disku

Zásadním problémem ve filesystému je určit, na kterém místě budou alokovány bloky souborů, aby co nejméně docházelo k fragmentaci[3]. Fragmentace je jev, kdy soubor není uložen v souvislých blocích. Fragmentace značně zpomaluje přístup k souborům

[1] Algoritmus prohledávání bitmapy funguje tak, že máme předpočítanou tabulku o velikosti 256 — cyklicky bereme každý byte bitmapy, ten použijeme jako index do tabulky, a bity výsledného bytu uloženého v tabulce říkají, zda je fragment dané velikosti v bytu dostupný. Odtud plyne omezení na velikost fragmentu jako osmina bloku.

[2] Filesystém bohužel v době zápisu bloku neví, jaká bude celková velikost souboru. Tento problém je vyřešen například v systému OS/2, kde uživatelský program může při vytváření souboru specifikovat jeho délku, pokud ji zná. Filesystém si pak na soubor vyhradí místo přesné velikosti.

[3] Fragmentace nastává na všech filesystémech a nemá nic společného s fragmenty na UFS, o kterých jsem se zmiňoval v předchozí kapitole.

— například pokud máme disk s přenosovou rychlostí 10MB/s a dobou přesunu hlavy 10ms, tak pokud bude soubor uložen ve fragmentech o velikosti 100kB, bude načítán dvakrát pomaleji. Neexistuje žádná teorie, která by se problémem alokace bloků na disku zabývala — všechny algoritmy jsou v podstatě empiricky vyzkoušené. Značnou nevýhodou při vyrábění algoritmu pro alokaci místa je, že špatnost nebo dobrost se pozná zpravidla až po několika měsících provozu — v podobě veliké nebo malé fragmentace. Navíc v různých použitích operačního systému jsou vyráběny soubory jiné velikosti a jsou mazány a vyráběny v jiném pořadí, proto algoritmus, který se osvědčí v jednom druhu zátěže, může zcela selhat v jiném druhu. Obecně platí, že na fragmentaci má velmi špatný vliv používání filesystému delší dobu ve stavu, kdy je téměř celý zaplněn[4]. Když už k zaplnění dojde, pro snížení fragmentace by měly být odmazány ty soubory, které byly zapsány naposledy a které zaplnění způsobily. Velmi nepříjemné je také, pokud jsou současně zapisovány dva dlouhé soubory v jednom adresáři. I když se filesystémy snaží tuto situaci alespoň trochu řešit, mnohdy výsledek dopadá tak, že jsou soubory na disku proházené jeden mezi druhým s velmi malými fragmenty, což rychlost čtení sníží na polovinu.

Kvůli nemožnosti odladění nebo rychlého vyzkoušení se algoritmy alokace příliš nemění. Bylo empiricky ověřeno, že existující algoritmy mají rozumné výsledky ve většině běžných použití, a nikdo tyto algoritmy nemá odvahu měnit, neboť by to mohlo přinést zhoršení, na které by se příliš pozdě a velmi těžko přišlo.

Na linuxovém filesystému Ext2 je algoritmus alokace následující: nejdříve se určí blok, poblíž něhož chceme alokovat (nazývaný goal). V případě, že soubor zatím nemá žádné bloky, goal je roven začátku skupiny. V případě, že soubor nějaké bloky má, je goal blok za posledním existujícím blokem souboru. Pak se zavolá funkce pro alokaci (`ext2_new_block`), která má za cíl alokovat blok poblíž goalu. Funkce pracuje následovně — pokud je goal volný, vrátí rovnou ten. Pokud je volný blok s číslem větším než goal a menším než (goal + 63) & ~ 63, je alokován tento blok. Pak se v bitmapě napravo od goalu hledá volný byte (čili 8 volných bloků), pokud je nalezen, alokuje se první blok na začátku tohoto souvislého volného místa. Pokud se v bitmapě nenalézá žádný volný byte, začne se hledat po jednotlivých bitech — najde se první volný blok napravo od goalu, a ten se alokuje. Pokud se ani takový blok nenajde, alokace pokračuje od začátku v další skupině (opět — nejdříve se hledá byte, pak se hledá bit, pak se přeskočí do další skupiny). Algoritmus nikdy nealokuje blok před goalem. Jediná možnost, kdy začne prohlížet bity před goalem, je, pokud už prošel celý disk, nic volného na něm nenalezl a vrátil se opět do výchozí skupiny.

Aby se zabránilo volání funkce alokátoru (která je celkem náročná) při každém zapsaném bloku a aby se zabránilo střídání bloků, pokud budeme současně zapisovat dva soubory, dělá Linux prealokaci. Alokační rutina nealokuje jeden blok, ale alokuje více souvislých bloků za alokovaným blokem, pokud se za ním nějaké volné bloky nacházejí. Implicitní hodnota je 8 předalokovaných bloků (to je dost málo — pokud budeme

[4] Například moje HPFS partition o velikosti 1,7G, která byla zaplněna na 93 – 97 procent po dobu čtyř let, má již průměrnou velikost fragmentu 38kB. Jiná partition, která byla podobně zaplněna, ale byla používána pro uložení operačního systému a nebylo na ni moc zapisováno, má průměrnou velikost fragmentu 74kB.

současně zapisovat dva soubory, budou se nám střídat po osmi blocích, což je celkem nepříjemné). Tuto hodnotu je možno změnit v superbloku, nicméně standardní programy `mke2fs` a `tune2fs` její nastavení neumožňují — člověk si tedy bude muset sám editovat obsah superbloku na disku. Když je soubor zavřen, jsou zbývající předalokované bloky uvolněny.

Algoritmus alokace byl vyvinut po zkušenostech s předchozími linuxovými filesystémy (Minix, Ext, XiaFS) a v praxi se ukázal jako kvalitní. Cílem algoritmu není a ani nesmí být „co nejméně zfragmentovat právě zapisovaný soubor“[5]. Můžeme si všimnout některých základních vlastností tohoto algoritmu.

- Prohledávání pouze dopředu — ačkoli pro minimalizaci pohybu diskové hlavy by mohlo být vhodnější alokovat blízký blok vzadu než vzdálený blok vpředu, algoritmus bloky vzadu nealokuje. Sám jsem alokaci směrem dozadu napsal při psaní ovladače filesystému HPFS a výsledek nebyl moc dobrý — soubor se rozlézal dopředu i dozadu a vzdálenost mezi fragmenty byla v důsledku toho ještě větší, než kdyby se prohledávalo jen dopředu.
- Vyhledávání celého volného bytu — to fragmentaci výrazně omezuje — soubor má pak aspoň osm souvislých bloků. Například filesystém XiaFS tohle nedělal, hledal jen následující volný bit a fragmentoval se značně. Nemá cenu si myslet, že pokud místo bytu budeme vyhledávat slovo, dvojslovo nebo větší souvislý úsek, tak fragmentaci ještě omezíme. Tím bychom omezili fragmentaci toho jednoho souboru, ale došlo by k závažnějšímu jevu — fragmentaci volného místa na disku. Na disku by pak zůstávalo spousta malých kousků volného místa, které jsme přeskočili ve snaze nezfragmentovat soubor, a jednoho dne pak zjistíme, že tyto kousíčky jsou jediné volné místo, které na disku zbylo. Vyhledávání delších úseků jsem také kdysi napsal a nebylo to dobré.
- Snaha zaplnit skupinu — pokud se ve skupině nevyskytuje volný byte, vyhledá se jakýkoli volný bit ve skupině. To sice značně zfragmentuje soubor, ale zabraňuje to fragmentaci volného místa, neboť skupiny jsou zcela zaplňovány. Kdyby se místo toho hledal volný byte v jiné skupině, došlo by po nějaké době k situaci, že v žádné skupině žádný volný byte není a po disku je spousta malých volných kousíčků.

Alokátor na FreeBSD UFS (funkce `ffs_alloc`) pracuje podobně — také se snaží blok alokovat blízko předchozího bloku souboru, ale má od Linuxu rozdíly. V bitmapě se nehledá osm souvislých bloků, ale pouze jeden blok (bloky jsou ovšem větší než fragmenty; v bitmapě každý bit odpovídá jednomu fragmentu). Když blok není nalezen vpředu ve skupině, hledá se od začátku skupiny. Pokud je celá skupina plná, hledá se další skupina pomocí tzv. „kvadratického rehashování“. Kvadratické rehashování spočívá v tom, že se filesystém podívá na následující skupinu, pokud je plná, tak na skupinu o dvě vzdálenou, pokud je plná, tak na skupinu o čtyři vzdálenou, pak o osm skupin dále atd. Tím se zajišťuje lepší distribuovanost dat po disku a omezuje se situace, kdy se dva soubory

[5] Algoritmus fungující podle tohoto cíle vypadá tak, že nalezne nejdelší volný úsek na disku a na jeho začátek začne zapisovat. Je zřejmé, že po delším provozu by na disku byly všechny velké úseky vyčerpány a byla by tam spousta malých kousíčků, což by fragmentaci mnohonásobně zvyšovalo.

budou vzájemně míchat mezi sebou. Další zvláštností alokátoru je, že pokud je v souboru alokován první nepřímý blok nebo pokud je alokován blok v nastaveném intervalu, tak přeskočí do další skupiny (přesněji: najde se další skupina, ve které je nadprůměrný počet volných bloků). Na první pohled to způsobuje fragmentaci souboru. Na druhou stranu to však umožňuje malé soubory ukládat ve skupině s jejich inodami a adresáři, neboť velké soubory tuto skupinu nezaplní.

Alokátor má rovněž optimalizace pro přístup k disku — může znát geometrii disku a snaží se pak bloky alokovat ve stejném válci. V případě, že máme disk, který se točí rychleji, než je počítač schopen odebírat data, může alokátor občas pár sektorů přeskočit, aby se data vždy četla přesně pod diskovou hlavou a nemuselo se čekat na jednu otáčku disku. Obě tyto optimalizace pocházejí ze starých dob, na moderních discích nemají absolutně žádný význam a je třeba je nechat vypnuté — současné disky svoji fyzickou geometrii skrývají, někdy nemají ani stejný počet sektorů na všech stopách, a pokud vůbec adresaci pomocí trojice (stopa/hlava/sektor) umožňují, přepočítávají tuto adresu na svoji interní reprezentaci. Vynechávání sektorů, aby se data četla rovnou pod hlavou, je zbytečné a zpomalující, neboť disky mají velikou read-ahead cache.

Jak již bylo řečeno, UFS může rozdělit blok na menší fragmenty. Alokátor má i funkce na alokaci fragmentů. V případě rozšiřování souboru jsou fragmenty přemisťovány do větších fragmentů nebo do bloku.

FreeBSD dělá tzv. clusterování zápisu. Zapisovaná data jsou skládána do bloků o velikosti až 128kB, tyto bloky jsou pak pomocí jednoho požadavku na blokové zařízení zapsány. Při skládání dat do clusteru filesystém dělá přemisťování nesouvislých bloků do souvislé oblasti. Je otázkou, zda je to dobré, nebo ne — zapisování dat po velkých blocích jistě pomůže zvýšit rychlost, nicméně přemisťování bloků je zase značně zpomalující.

9.4. Zajišťování konzistence filesystému v případě výpadku

Pro zvýšení výkonu mají filesystémy asynchronní zápis — t.j. data nezapisují na disk okamžitě, ale drží si je v bufferové cachi a počkají, než se akumuluje dostatečné množství dat nebo než uplyne nějaká dlouhá doba (typicky v řádu desítek sekund až minut; je možno to v systému nastavit) — teprve pak zapíší. Po výpadku proudu nebo pádu operačního systému se filesystém nachází v nekonzistentním stavu. Některé sektory byly zapsány, některé nebyly, a pokud bychom s takovým filesystémem pracovali, docházelo by k závažným chybám (například pokud byl zapsán soubor, ale nebyla ještě zasána bitmapa alokovaných bloků, bude soubor sice existovat a obsahovat data, ale v náhodný okamžik bude přepsán jiným souborem). Pokud se při startu filesystém nachází v nekonzistentním stavu, bude automaticky zkontrolován programem `fsck`, který nekonzistence opraví. To s sebou přináší problémy:

- Je to značně pomalé. Běžně to trvá několik minut; na velkých discích s mnoha malými soubory to může být až několik hodin.
- I když program `fsck` zkontroluje a do konzistentního stavu uvede všechny řídící struktury, nemůže do konzistentního stavu uvést samotná data, neboť neví, jaká data soubory mohou obsahovat. Kromě nesmyslných dat v souborech to přináší i ohrožení **bezpečnosti systému**. Představme si situaci, kdy uživatel 1 vyrobí soubor, uloží

do něj svoje tajná data a pak soubor smaže. Uživatel 2 vyrobí soubor, filesystém umístí soubor na stejné místo, kde byl soubor uživatele 1. K přerušení proudu dojde v okamžiku, kdy byly zapsány alokační informace souboru, ale nebyla ještě zapsána samotná data. Po novém startu a provedení kontroly filesytému bude uživatel 2 ve svém souboru vidět tajná data uživatele 1. Tato chyba se dá zneužít i úmyslně, stačí, když útočník vyrobí soubor tak velký, aby ho jádro ještě nezačalo zapisovat, pak počká, než se zapíšou řídící informace, ale dokud se ještě nezapíšou data, vypne proud nebo shodí systém. Po novém startu systému může útočník číst data, která mu nepatří.

Používají se následující metody, které zaručují vyšší konzistenci dat:

- **Synchronní zápis** — nejstarší a nejsnadněji implementovatelná metoda. Umí ji Linux i FreeBSD. Data se na disk fyzicky zapisují okamžitě v systémových voláních `write`, `mkdir` apod. Vede to bohužel k několikanásobnému zpomalení diskových operací, proto se tato metoda téměř nepoužívá. Synchronní zápis může vést k nekonzistentnímu stavu filesystému, nicméně tento nekonzistentní stav nezpůsobí žádné ztráty dat. Je možné, že se budou na filesystému objevovat ztracené bloky či inody, ale není možná situace, kdy soubor bude existovat, ale nebude zapsán v bitmapě, což by vedlo k náhodnému přepsání souboru. Ztracené bloky a inody je možno uvolnit zkontrolováním celého filesystému pomocí `fsck`. Synchronní zápis v Linuxu ani FreeBSD nezaručuje bezpečnost dat — v případě pádu v nevhodnou dobu může uživatel číst data, která mu nenáleží. Synchronní zápis by bylo možno implementovat tak, aby bezpečnost dat zaručoval, ale v Linuxu ani FreeBSD to tak není.
- **Soft updates** — metoda implementovaná pouze na FreeBSD. Bufferová cache obsahuje závislosti mezi jednotlivými buffery — závislost je typu, že buffer X musí být zapsán dříve než buffer Y. Kernel thread starající se o zápis modifikovaných bufferů zapisuje buffery tak, aby tyto závislosti nebyly porušeny. Kód filesystému vždy při značení špinavých bufferů popisuje, jaké závislosti mezi nimi jsou. Například při vytváření souboru je řečeno, že položka v adresáři musí být zapsána později než inoda a že položka v adresáři musí být zapsána později než bitmapa inod. Tím se zajistí, že ať systém spadne v jakémkoli okamžiku, bude adresář konzistentní. Sice může vzniknout ztracená inoda, na kterou nikdo neukazuje, ale nemůže vzniknout situace, kdy adresář ukazuje na nesmyslnou inodu. Při zápisu dat do souboru se rovněž vytvářejí tato pravidla: ukazatel na blok musí být zapsán později než data v tomto bloku a ukazatel na blok musí být zapsán později než bitmapa. Může tedy vzniknout ztracený blok, ale nemůže vzniknout ukazatel na nealokovaný blok nebo ukazatel na blok obsahující náhodná data. Bezpečnost filesystému tedy **je** zajištěna.
  Filesystém musí zvlášť dávat pozor na cyklické závislosti. Například pokud je přesunut soubor z adresáře A do adresáře B, vznikne závislost, že adresář A je třeba zapsat dříve než adresář B (neboť ztracený soubor je menší škoda, než soubor s refcount 1, na který se odkazují dva adresáře). Pokud je přesunut další soubor z adresáře B do adresáře A, vznikne opačná závislost: adresář B je třeba zapsat dříve než adresář A. Kód musí tuto situaci detekovat, a pokud nastane, tak buffer obsahující A zdvojit, původní kopii znepřístupnit pro vyšší vrstvy, do nové kopie nechat zapisovat změny a vyrobit závislost říkající, že: nejdříve se zapíše původní buffer adresáře A, pak se

zapíše buffer adresáře B a pak se zapíše nový buffer adresáře A.

- **Žurnálování** — Žurnálování je metoda, která umožňuje udržet po výpadku proudu zcela konzistentní stav filesystému. Není třeba provádět kontrolu filesystému a nevznikají ztracené bloky nebo inody jako u soft updates. Na filesystému je vyčleněna oblast (veliká obvykle pár megabytů) nazývaná žurnál. Ovladač filesystému musí změny provádět v tzv. transakcích. Transakce je několik modifikací dat, které převádějí filesystém z jednoho konzistentního stavu do druhého konzistentního stavu. Například při vytváření souboru se vyrobí transakce, do které náležejí následující operace: zapsání položky adresáře, zapsání inody, zapsání bitu v bitmapě alokovaných inod, zmenšení počítadla volných inod v superbloku. Operace náležející transakci se nezapisují na disk, ale nejdříve se zapisují do žurnálu. Až je celá transakce v žurnálu zapsaná, zapíše se do něj speciální značka — commit transakce. Poté se buffery obsahující jednotlivé modifikace mohou zapisovat na disk jako při asynchronním zápisu. Do žurnálu se zpravidla zapisuje pořád dopředu, až dojde k jeho zaplnění, začne se zapisovat opět od začátku (při této operaci je třeba zapsat na příslušná místa na disku všechny buffery obsahující modifikovaná data). Při mountu filesystému se kontroluje žurnál. Pokud je v něm nalezena transakce včetně commit značky, tak se všechny operace náležející této transakci zapíší na příslušná místa na disku. Pokud je v žurnálu nalezena neúplná transakce bez commitu, ignoruje se.
  Žurnálování zajišťuje absolutní konzistenci filesystému po výpadku. Pokud počítač spadl během zápisu transakce do žurnálu, je transakce v žurnálu neúplná a bude při příštím startu ignorována. V takovém případě máme garantováno, že ještě nedošlo k zápisu žádných dat na disk mimo žurnál. Pokud celý zápis transakce do žurnálu proběhl (včetně commit značky) a počítač spadl během zápisu dat na disk, bude při příštím startu celá transakce z žurnálu rekonstruována, čímž budou nekompletní data na disku přepsána. Ať spadne počítač v jakémkoli okamžiku, data budou konzistentní. Aby žurnálování mohlo správně fungovat, předpokládá se, že disk je schopen provést tzv. atomický zápis sektoru — t.j. v sektoru se po výpadku budou nacházet buď úplná stará, nebo úplná nová data. Pokud dojde k výpadku proudu během zápisu sektoru, disk musí být schopen tento zápis dokončit. Současné disky to umějí. Schopnost atomického zápisu sektoru nemají například diskety (pokud disketovou mechaniku vypnete během zápisu dat, výsledkem bude neúplně zapsaný sektor, který nejspíše bude produkovat CRC chybu), proto používání žurnálového filesystému na disketách nemá smysl.
  Do žurnálu se zpravidla zapisují pouze metadata (t.j. řídící informace filesystému) a nikoli samotná data obsažená v souborech. Důvodem je rychlost — při žurnálování je potřeba na disk zapsat dvakrát více dat než při normálním zápisu a zapisování celých souborů by proces zápisu dvakrát zpomalilo. Pokud se data do žurnálu nezapisují, je třeba zajistit jejich konzistenci, aby nevznikl výše popsaný bezpečnostní problém: metadata jsou zapsaná, data nejsou zapsaná, uživatel po pádu bude moct číst cizí smazané soubory. Obecně platí, že před zapsáním commit značky je potřeba zapsat všechna nově vzniklá data. Musí se také udržovat konzistence mezi datovými a metadatovými buffery, což logiku bufferové cache komplikuje (například není možné zapsat data do bloku, ve kterém dříve byla metadata, a v žurnálu je ještě nějaká transakce pracující s těmito metadaty).

Neexistuje žurnálový filesystém pro FreeBSD. Pro Linux existuje několik žurnálových filesystémů:

Ext3 — tento filesystém má zcela shodnou strukturu s filesystémem Ext2 až na to, že provádí žurnálování. Existující filesystém Ext2 je možno přetvořit na Ext3 pouhým přidáním speciálního souboru obsahujícího žurnál. Ext3 je možno zpětně namountovat jako Ext2 (v takovém případě se neprovede obnovení žurnálu — proto před tím nesmělo dojít k pádu). Ext3 může pracovat ve třech režimech: **Unordered data** — nebude udržováno pořadí mezi daty a metadaty, takže zde existuje bezpečnostní problém; **Ordered data** — bude zajištěno, že data budou zapsána dříve než commit transakce, takže je filesystém bezpečný; **Journal data** — data budou zapisována do žurnálu. To dvakrát zpomalí rychlost zápisu dat, ale způsobí, že bude udržováno i pořadí jednotlivých volání funkce `write`. Doporučená metoda je ordered data, což zajišťuje bezpečnost a nezpomaluje tolik jako žurnálovaná data.

Ext3 se nachází v jádrech 2.4 i 2.5 (je možno stáhnou patch Ext3 i pro jádra 2.2), ale z komentářů v kódu je vidět, že ještě úplně odladěné není — může se vyskytovat špatné předpovídání velikosti transakce, což může vést k přeplnění žurnálu, synchronizace metadat a dat také není absolutně v pořádku. Tyto chyby se v běžných případech moc nevyskytují.

ReiserFS — tento filesystém původně nebyl žurnálovaný; žurnál byl do něho dodělán až později. ReiserFS nezaručuje pořadí zápisu dat a metadat (a z kódu je patrné, že programátoři na tento problém vůbec nepomýšleli), proto na něm existuje bezpečnostní problém.

JFS — tento filesystém byl vyvinut firmou IBM a je používán na jejích operačních systémech AIX a OS/2. Jeho kód byl uvolněn pod GNU General Public License a zařazen do experimentálního Linuxu 2.5. Vzhledem k tomu, že se nachází pouze v experimentálních jádrech, není vhodný pro běžné použití. Bezpečnostní problém s pořadím zápisu dat a metadat se tam programátoři snažili řešit, ale v současné verzi to nefunguje. Dá se předpokládat, že před vydáním stabilního jádra bude tento problém opraven.

- Existují i další metody, které zajišťují podobnou funkčnost jako žurnálování, ale jejichž princip funkce je zcela jiný. Existují pouze v experimentálních filesystémech, do jádra se nedostaly.

  Fázový strom — na filesystém je možno pohlížet jako na strom ukazatelů: superblok ukazuje na kořenový adresář a na bitmapy, adresář ukazuje na inody, inody ukazují na podadresáře a tak dál. Metoda fázového stromu funguje tak, že všechny zápisy se dělají na nealokovaná místa na disku. Proto tyto zápisy nemohou ohrozit konzistenci dat. Když se například má zapsat inoda, zapíše se na místo, které je v bitmapě inod nealokované. Pak se na další nealokované místo zapíše adresář, ve kterém se inoda nacházela (a v tomto adresáři se modifikuje ukazatel na inodu na její nové místo). Pak se opět na nealokované místo zapíše inoda toho adresáře, její nadadresář a tak dále až ke kořeni. Na nealokovaná místa se zapíší i nové bitmapy, které tyto změny popisují. Nakonec se zapíše superblok s novým ukazatelem na nový kořenový adresář a na nové bitmapy, čímž se během jedné operace zápisu filesystém dostane z jednoho konzistentního stavu do druhého. Tato metoda vede k zápisu značného množství dat, je pomalá, a proto se moc nepoužívá.

Crash counts — filesystém má počítadlo pádů (o velikosti například 16 bitů) a tabulku, kde pro každou hodnotu počítadla pádů je počítadlo transakcí. Po mountu filesystém na disku zvýší počítadlo pádů o 1 (ale v paměti si nechá staré), načte tabulku počtu transakcí a v paměti (na disk nezapisuje) zvýší o jedna hodnotu na pozici počítadla pádů. Při odmountování filesystému je počítadlo pádů na disku opět zmenšeno o 1. Každý pointer na disku má kromě adresy dvě položky — crash count (`cc`) a transaction count (`txc`). Pointer je platný, pokud `crash_count_table[cc] - txc >= 0`. Když filesystém zapisuje nějaká metadata, zapisuje k pointerům aktuální crash count a transaction count. Pokud systém spadne, bude při dalším startu pracovat s crash count větším o 1 (načteným z disku) a všechny pointery vyrobené do doby pádu budou považovány za neplatné. Když se zapíše tabulka počítadel transakcí na disk, začnou být náhle všechny dříve zapsané pointery platné. Pak se v paměti zvýší počítadlo transakcí o 1 a mohou se zapisovat nová data. Některé struktury, které nemají povahu pointerů (např. bitmapy), je nutno na disku ukládat dvakrát a pro tuto dvojici mít `cc` a `txc` určující, která z bitmap je platná. Při zápisu do bitmapy se zkontroluje, zda `cc` a `txc` bitmapy jsou aktuální hodnoty, se kterými se pracuje — pokud ano, zápis se provede (a v případě pádu bude platná ta druhá bitmapa, do které se nezapisovalo). V případě, že `cc` a `txc` bitmapy nejsou aktuální, musí dojít ke zkopírování platné bitmapy do druhé a nastavení `cc` a `txc` dané dvojice bitmap na aktuální hodnoty. Tato metoda má tu nevýhodu, že umožňuje přežít pouze pevný počet pádů (v případě 16bitového crash count je to 65536) — až crash count dojde na horní hranici, je třeba stejně celý filesystém projít, zkontrolovat, vymazat neplatné pointery a crash count vrátit zpátky na nulu.

9.5. Nedostatky unixového filesystému

Unixový filesystém vznikl před více než 30 lety, a proto má některé nedostatky, které se v novějších filesystémech podařilo odstranit.

- Předalokované místo na inody — je zbytečné umisťovat inody pouze na pevná předalokovaná místa. Volné inody pak buď zabírají zbytečně mnoho místa, nebo dochází k opačnému jevu, kdy inody docházejí. Lepší filesystémy jsou schopny alokovat inody kdekoli na disku.
- Popis alokace souboru pomocí přímých a nepřímých bloků — tato metoda umožňuje vyhledat libovolnou část souboru v konstantním čase (na rozdíl třeba od FAT filesystému, kde k nalezení konce souboru je třeba sekvenčně projít ukazatele na všechny jeho bloky). Nepřímé bloky však zabírají spoustu místa, jejich čtení zabírá čas, a i když je přístup k libovolné části souboru v konstantním čase, k tomuto přístupu potřebujeme až tři přístupy na disk. Lepší je využít předpokladu, že fragmentace souborů je malá, a alokační informace ukládat do dvojic (blok na disku, počet souvislých bloků). Tyto dvojice je pak možno skládat do b-stromu, lineárního seznamu (z hlediska teoretické informatiky je lineární seznam špatný, nicméně vzhledem k tomu, že soubory moc zfragmentované nebývají, není tato struktura špatná) nebo na ně ukazovat indirect bloky různé úrovně.
- Adresáře jako lineární seznamy — vyhledávání v nich je pomalé, pokud adresář obsa-

huje spoustu souborů. Některé filesystémy ukládají adresáře do b-stromu, takže jsou schopny libovolný soubor nalézt v logaritmickém čase.

- Informace jsou v inodách a ne v adresáři — pokud na Unixu napíšeme `ls -la`, bude tato operace trvat poměrně dlouho, protože se pro každý soubor adresáře bude muset přečíst jeho inoda. Některé filesystémy mají informaci o souborech uloženou rovnou v adresářových položkách, takže daná operace na nich bude trvat výrazně menší dobu. Pokud jsou informace o souboru v adresářových položkách, neumožňuje to použít hardlinky; na druhou stranu uživatelé hardlinky moc často nepoužívají a používají mnohem raději symlinky, neboť hardlinky nejsou vidět ve výpisech adresáře a vedou ke zmatenosti uživatele.

Existují novější filesystémy, které se tyto problémy snaží nějak řešit. Na FreeBSD není v současné době jiný použitelný diskový filesystém než UFS. Na Linuxu existují kromě původního filesystému Ext2 i alternativy:

- **Ext3** — tento filesystém má zcela stejnou strukturu jako Ext2, proto žádné zrychlení operací nepřináší. Ext3 se nachází v jádrech 2.4 a 2.5.
- **ReiserFS** — Filesystém ReiserFS je optimalizován pro ukládání malých souborů. Filesystém ukládá do stromu nejen obsah adresáře, ale i jednotlivé inody a malé soubory. Dokáže do jednoho bloku uložit několik malých souborů, proto je vhodný na proxy cache, news nebo mail servery, nebo jiné aplikace, které potřebují pracovat s velkým množstvím malých souborů. Na druhou stranu, prohledávání stromu při vyhledávání inody zpomaluje, proto je tento filesystém při některých typech zátěže pomalejší než Ext2.
- **JFS** — Filesystém JFS má adresáře ve tvaru stromu, popis bloků náležících souboru rovněž ve tvaru stromu, a má dynamicky alokované místo na inody. Tento filesystém se nachází pouze v experimentálních jádrech 2.5.

9.6. Další filesystémy v jádrech

Na Linuxu existuje i větší množství dalších filesystémů, které byly napsány kvůli kompatibilitě s jinými operačními systémy. Často k nim chybí potřebné programy pro vytvoření a zkontrolování filesystému a nemá je smysl používat, pokud nemáme na stejném počítači jiný operační systém, který je vyžaduje. Patří sem ADFS (Acorn disk file system — nativní filesystém systému RiscOS), AFFS (Amiga fast file system), BFS (UnixWare boot file system), EFS (filesystém starších systémů IRIX), FAT (filesystém DOSu a Windows 95/98/ME), HFS (filesystém na Macintoshi), HPFS (High-performance file system — filesystém používaný systémem OS/2), ISOFS (filesystém na CD), JFFS (Journaled flash file system — filesystém optimalizovaný na flash karty místo na disky), Minix (filesystém dnes již historického operačního systému Minix), NTFS (filesystém Windows NT/2000/XP, podpora v Linuxu je read-only), QNX4 (filesystém operačního systému QNX, v Linux read-only), SYSV (filesystém používaný na komerčních Unixech Xenix, SCO a Coherent), UDF (nový filesystém na CD), UFS (rodina filesystémů používaná na FreeBSD, NetBSD, OpenBSD, SunOS, NeXTstep; na některé typy tohoto filesystému Linux umožňuje zapisovat, na jiné ne), VXFS (nativní filesystém UnixWare; na Linuxu read-only).

Na FreeBSD je množství cizích filesystémů výrazně menší — FreeBSD podporuje FAT, Ext2 (čtení i zápis, ale ve spoustě verzí FreeBSD dost zabugovaný, kdysi mi to zničilo jeden blok v tabulce inod), HPFS (read-only), ISOFS, NTFS (read-only), UDF.

10. Virtuální paměť

Každý proces má svůj vlastní adresní prostor, který určuje jeho tabulka stránek. Pokud je stránka procesu v jeho tabulce stránek, proces na ni může okamžitě přistupovat. Při přístupu je virtuální adresa převedena na fyzickou adresu podle tabulky stránek. Pokud má systém nedostatek fyzické paměti, začne stránky zapisovat na disk do swapovací oblasti. Když je stránka zapsaná na disk, do tabulky stránek procesu je poznamenáno, že stránka je nepřístupná. Pokud proces na tuto stránku přistoupí, vyvolá se exception, který je zpracován operačním systémem. Při přijmutí exceptionu operační systém natáhne stránku z disku zpět do paměti (případně odswapuje jinou stránku na disk), nastaví v tabulce stránek procesu, že stránka je platná, a pustí dál proces, který výpadek stránky vyvolal. — tohle je asi základní myšlenka, která se za principem virtuální paměti skrývá a která se učí na kurzech operačních systémů. Skutečnost je však mnohem komplikovanější. Na systém virtuální paměti je kladeno větší množství požadavků:

- Sdílení kódu programů — pokud více uživatelů pustí tentýž program, je žádoucí, aby kód programu byl v paměti zaveden jen jednou.
- Load-on-demand — určitá data (například kód programů) je možno načíst z filesystému v případě potřeby. Program proto není třeba číst celý v době jeho spuštění, ale je načítán až v době běhu. Načteny jsou jen ty části programu, které jsou skutečně použity. Pokud dochází paměť, není třeba program ukládat do swap oblasti, stránky obsahující kód programu jsou prostě uvolněny, neboť je možno je kdykoli znovu načíst ze souboru.
- Mapování souborů — souvisí s load-on-demand a jeho implementace je stejná. V tomto případě se však nenačítá kód programu, ale libovolný datový soubor. Každý proces může požádat o namapování nějakého souboru do svého adresního prostoru. Pokud na příslušnou adresu přistupuje, přistupuje do souboru. Mapování se provádí pomocí syscallu `mmap`. Máme tři druhy mapování: pro čtení (do namapované oblasti nelze zapisovat, příznak `PROT_READ`), pro privátní zápis (do namapované oblasti lze zapisovat, tyto zápisy platí pouze pro daný proces, nezapisují se zpět do souboru, zapomenou se při zrušení mapování, příznaky `PROT_WRITE` a `MAP_PRIVATE`), pro sdílený zápis (do namapované oblasti lze zapisovat, zápisy jsou viditelné všemi ostatními procesy a jsou zapsány zpět do namapovaného souboru[1], příznaky `PROT_WRITE` a `MAP_SHARED`).
- Copy-on-write — technika copy-on-write se používá k efektivní implementaci syscallu `fork`. Já osobně považuji `fork` za nejhorší chybu návrhářů Unixu. Syscall `fork` spustí podproces daného procesu. Podproces i rodičovský proces pokračují v běhu od stejného místa. Jednoduchá implementace `fork` funguje tak, že celou datovou oblast procesu zkopíruje do nového procesu. To je velmi pomalé. Proto se k implementaci `fork` používá virtuální paměť. Při `fork` se starému procesu nastaví všechna mapování stránek read-only. Pak se tabulka stránek nakopíruje do nového procesu. Když některý proces do svých dat zapíše, dojde k exceptionu a systém v obslužné

[1] Podle standartu POSIX zápisy nemusí být vidět v souboru a v mapování jiných procesů, dokud se oblast neodmapuje nebo dokud se nezavolá syscall `msync`. Na většině systémů jsou zápisy vidět okamžitě, nicméně program, který na to spoléhá, je chybný a na starších systémech nemusí fungovat.

rutině tohoto exceptionu udělá kopii stránky a k této kopii nastaví přístup read-write. Procesy tak mají dojem, že běží každý ve svém vlastním adresním prostoru, ale ve skutečnosti jsou jejich data stále sdílená, dokud do nich nezapíší. Copy-on-write je rychlejší než kopírování celých procesů, ale stále je dost pomalé, neboť se musí kopírovat tabulka stránek. Nejhorší však je, že kopírování tabulky stránek je při pouštění jiného programu jako podprocesu zcela zbytečné. Všechny ne-unixové systémy (např. OS/2, Windows, VMS i MS-DOS) mají syscall `spawn` nebo jeho ekvivalent, který jako parametr dostane jméno programu a argumenty a pustí tenhle podproces. Pokud chceme pustit podproces na Unixu, musíme nejdříve rodičovský proces rozdvojit pomocí `fork` a poté zavolat v dětském podprocesu `exec`, což procesu smaže data a pustí místo něj specifikovaný program. Tabulka stránek se tak pracně zkopíruje jen na to, aby se okamžitě smazala pomocí `exec`[2]. Na některých Unixech (Linux i FreeBSD mezi ně patří) byl zaveden syscall `vfork`, který provede totéž co `fork` vyjma nastavování stránek read-only a kopírování tabulky. `vfork` také zablokuje rodičovský proces, dokud dětský podproces nezavolá `exec`. Předpokládá se, že dětský podproces provede ihned po `vfork` `exec`, čímž zavede nový program. Mezi `vfork` a `exec` nesmí dětský podproces modifikovat žádné proměnné, neboť tahle modifikace se může a nemusí přenést do rodičovského procesu. Kompilátor ovšem občas spontánně do zásobníkového rámu píše nějaké dočasné hodnoty, proto v rodičovském procesu po `vfork` přestávají platit hodnoty všech lokálních proměnných v aktuální funkci. Použití `vfork` je tedy poměrně komplikované.

- Sdílená paměť — procesy mohou sdílet paměť pomocí syscallů SYSV SHM. Toto rozhraní je značně těžkopádné, pro každý sdílený segment se musí vyrábět speciální klíč a ten pak distribuovat mezi procesy (pokud nebude vyroben unikátní klíč, ale bude použita pevná hodnota, tak program sestávající z více procesů sdílejících paměť nebude moci používat více uživatelů současně). Proto nová norma POSIX umožňuje sdílet paměť pomocí mapování souborů (funkce `shm_open`). Segment sdílené paměti se identifikuje pomocí řetězce a ne pomocí číselného klíče, což zabraňuje kolizi klíčů mezi jednotlivými programy nebo mezi více instancemi téhož programu.
- Cachování — je žádoucí, aby zbylá volná paměť byla použita jako disková cache. Dnešní počítače mají velké množství paměti, málokterý program celou paměť využije a cachování souborů v nepoužité paměti je zcela zásadní nutnost správy virtuální paměti. Kvalita cache výrazně určuje rychlost systému. Dnes již u virtuální paměti nejde ani tak o swapování (ke swapování dochází zřídka), ale právě o cachování souborů. Je třeba, aby nedošlo k tzv. vytrashování cache čtením nebo zápisem velkého souboru. Pokud budeme sekvenčně číst soubor větší než velikost paměti, není žá-

[2] Původní Unix běžel na PDP-7, které nemělo virtuální paměť, segmentaci ani ochranu paměti. V paměti mohl být zaveden vždy jen jeden proces, který běžel. Multitasking se dělal swapováním celých procesů na disk. V tomto prostředí je implementace syscallů `fork` a `exec` velmi jednoduchá — `fork` pouze odswapuje aktuální proces na disk, ale nechá ho běžet s novým PID; `exec` načte do paměti přes existující proces nový program. Proto ani není divu, že autoři původního Unixu `fork` a `exec` implementovali tak, jak jsou. Bohužel v systémech se segmentací nebo virtuální pamětí je tato implementace zoufale neefektivní.

doucí, aby se tento soubor ukládal do cache. To by vedlo k úplnému „přemazání“ všech údajů v cachi naposled načtenými stránkami souboru. Až bude soubor čten znovu od začátku, nebude cache užitečná; pravděpodobnost, že soubor bude čten znovu od konce (kde jsou nacachovaná data) je velmi malá.

- Balancování cachování a swapování — v určitých situacích je v systému zaveden veliký program, který neběží. Pak je žádoucí program odswapovat na disk a uvolněnou paměť používat jako cache. V jiných případech se zase stává, že běží jeden veliký program, který cache téměř nepotřebuje. Pak je žádoucí velikost cache stáhnout na minimum a veškerou paměť dát onomu programu.
- Spravedlivost vůči uživatelům — jeden uživatel by neměl mít možnost nekontrolovatelně vyswapovávat stránky ostatních uživatelů a zpomalovat jim tak běh jejich procesů. Současné systémy tenhle požadavek příliš nesplňují (mají jakousi podporu pomocí příkazu `ulimit`, FreeBSD je na tom lépe než Linux, ale k dokonalosti má tohle řešení hodně daleko). Jediný systém, na kterém je tento problém rozumně vyřešen, je VMS.

10.1. Historie virtuální paměti

První pokusy s něčím, co trochu připomínalo virtuální paměť, začaly v interpretech jazyka LISP. LISPové struktury CONS byly ukládány na disk. Při procházení pointerů bylo interpretem zjišťováno, zda pointer ukazuje na strukturu v paměti, nebo na disku, a struktura byla případně z disku nahrána. Procesory v té době neměly žádnou virtuální paměť, virtualizace se dělala pouze v rámci interpretu.

Prvním systémem, který měl komplexní podporu virtuální paměti, byl Multics. Multics splňoval téměř všechny požadavky na virtuální paměť: sdílení kódu programu, používání paměti jako souborové cache, mapování souborů do paměti, jednotný pohled na cachované stránky a na stránky alokované procesy. V Multicsu se pro práci se soubory používalo výhradně mapování — klasické unixové syscally `read` a `write` tam neexistovaly. Mapování je efektivnější než `read` a `write`, neboť při něm nedochází ke kopírování dat. Pro výměnu stránek používal Multics prostý hodinový algoritmus (popis algoritmu viz níže). Praxe však ukázala, že to nefungovalo příliš efektivně — jednotný pohled na alokované stránky a na cache sice umožnil efektivní cachování, ale způsoboval, že pokud někdo přečetl veliký soubor, všem ostatním uživatelům byly stránky odswapovány. V dobách, kdy byl Multics používán, byly na počítače kladeny mnohem větší požadavky než dnes; počítačů bylo málo a uživatelů hodně. Na Multicsu velmi často docházelo k tzv. trashování. Tento jev nastává, když množství paměti, na kterou aktivní procesy přistupují, je výrazně větší než množství fyzické paměti — skoro každý přístup do paměti pak způsobí page-fault a načtení stránky z disku, během operace disku je naschedulován jiný proces, ten ovšem také okamžitě způsobí page-fault, zařadí požadavek na čtení stránky do fronty a čeká, je naschedulován další proces, který udělá další fault a tak dále ... výsledek je takový, že fronta požadavků na disk je zaplněna a žádný proces se téměř nehne z místa. Kdyby byly procesy pouštěny sekvenčně po sobě, doběhnou o několik řádů rychleji, než když jsou puštěny paralelně. Je paradoxní, že Linux 2.2 používá rovněž hodinový algoritmus k výměně stránek podobně jako Multics, a přitom si na pomalost Linuxu nikdo tolik

nestěžuje — je to dáno tím, že dnes jsou na počítače kladeny výrazně nižší nároky. Kdyby Linux měl podporovat přístup 40 uživatelů na stroji s 4M ram, byl by zcela nepoužitelný.

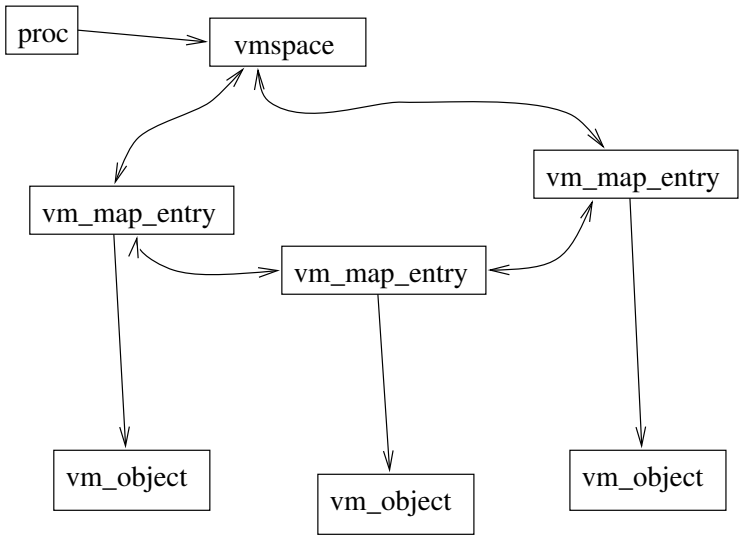
Neúspěch Multicsu a problém trashování vedl k jednoduchému řešení — nepoužívat virtuální paměť vůbec. Tak byl implementován Unix. Unix nedělal stránkování a swapoval celé procesy, původní verze Unixu mohly mít v paměti zaveden jen jeden proces, novější verze mohly mít v paměti více procesů. I když je swapování procesů výrazně primitivnější činnost než stránkování, v určitých situacích se chová lépe. Při swapování procesů nedochází k trashování; pokud je systém přetížen mnoha procesy, má sice velmi dlouhou odezvu, ale procesy běží a konají nějakou práci. Naproti tomu při trashování procesy téměř neběží. Problematikou cache se v Unixu příliš nezabývali — systém měl malou fixní část paměti předalokovanou na buffery a to bylo všechno. V té době ani cache nebyla potřeba — pokud byla v systému nějaká volná paměť, určitě se našel nějaký uživatel, který by se rád přihlásil a paměť využil, takže plýtvání pamětí na cache nemělo smysl. Spousta Unixů (OpenBSD, Irix, Solaris[3]) bohužel má bufferovou cache fixní velikosti dodnes. V praxi to pak vypadá tak, že systém má třeba 2/3 paměti volné, ale nepoužije ji jako cache a soubory znovu a znovu pomalu čte z disku.

Poučení z pomalosti Multicsu si vzali návrháři VMS a udělali virtuální paměť tak, že k samovolnému vytrashování nemůže dojít. VMS používá kombinaci stránkování a swapování celých procesů. Na VMS má každý proces nastavenou tzv. „working set“. Je to množství paměti, které má proces namapované — t.j. množství položek tabulky stránek, které mají bit PRESENT nastaven a ukazují na stránku. Pokud chce proces přistupovat na více paměti, než je jeho working set, systém mu automaticky nějaké stránky odmapuje a umístí je do clean nebo modified listu, aby working set byla zachována. Namapovaná paměť procesu nikdy neklesne pod jeho working set. Když v systému dochází paměť, systém zapisuje stránky z modified listu na disk nebo uvolňuje stránky z clean listu. Pokud už se takto žádnou paměť nepodaří uvolnit, systém začne swapovat celé procesy. Až je proces později naswapován, jsou nataženy všechny jeho stránky, které byly namapovány před odswapováním. Tento přístup má výhody virtuální paměti bez nebezpečí trashování — protože množství stránek procesu zavedeného v paměti není nikdy menší než jeho working set, proces může aspoň chvíli po naswapování běžet, aniž by produkoval další page-faulty a způsoboval trashování. Pokud je systém pod malou zátěží, chová se jako systém s virtuální pamětí a stránkováním. Pokud je pod velkou zátěží, začne se chovat jako systém bez virtuální paměti se swapováním celých procesů, což je v takovém případě efektivnější. Tento systém je také spravedlivý vůči uživatelům. Pokud jeden uživatel začne používat enormní množství paměti, bude jeho proces odswapovávat a naswapovávat jenom svoje vlastní stránky a neovlivní to nijak paměť ostatních uživatelů. Původní VMS nemělo žádnou cache (neboť v té době nebyla cache potřeba), nové verze ji už mají, ale není do zmíněného systému virtuální paměti moc dobře integrovaná a není tam stejný pohled na stránky nacachované a namapované. Navzdory tomu, že správa virtuální paměti na VMS zabraňuje trashovnání a je spravedlivá vůči uživatelům, současné operační systémy tyto principy nepoužívají, neboť zátěž na ně kladená je výrazně nižší.

[3] nejsem si jist, zda to platí i pro nejnovější verze

10.2. Struktury virtuální paměti na FreeBSD

Každá fyzická stránka paměti je popsána strukturou `struct vm_page`. Každá stránka náleží do právě jednoho objektu (`struct vm_object`). Objekty tvoří stromy — každý objekt může mít jeden „backing object“. Některé objekty backing objekt mít nemusí. Každému cachovanému souboru v paměti náleží jeden objekt. Každému procesu náleží objekt, který popisuje paměť alokovanou tímto procesem. Každému segmentu sdílené paměti náleží také jeden objekt. Každý objekt má „backing store“ — je to oblast, kam se budou stránky zapisovat a ze které se budou zpět natahovat. Pro objekty náležící souborům je backing store daný soubor; pro objekty paměti procesů nebo sdílené paměti je backing store swapová oblast. Každý proces má jednu `struct vmspace`, která popisuje jeho virtuální adresní oblast. `struct vmspace` má kruhový seznam struktur `struct vm_map_entry` — každá `vm_map_entry` odpovídá jednomu segmentu namapovanému pomocí `mmap`. `vm_map_entry` obsahuje ukazatel na `vm_object` a adresu, na kterou je tento objekt namapován.



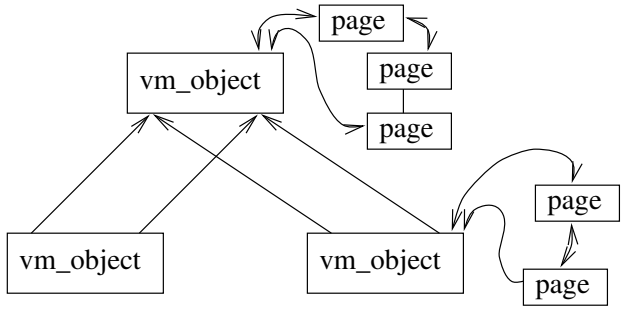
Popis virtuálního adresního prostoru

Struktura `struct vmspace` má položku `struct pmap`, která obsahuje informace pro virtuální prostor procesu závislé na architektuře. Každá stránka má kruhový seznam struktur `struct pv_entry`. Každá `pv_entry` odpovídá jednomu mapování této stránky. `pv_entry`[4]je struktura závislá na architektuře. S mapováním se manipuluje pomocí obecných funkcí, jako například `void pmap_enter(struct pmap *pmap, unsigned long vm_offset, struct vm_page *page, vm_prot_t prot, boolean_t wired)`. Tato funkce namapuje stránku do adresního prostoru procesu. První parametr je adresní prostor, druhý parametr je adresa v tomto prostoru, třetí parametr je stránka, čtvrtý parametr jsou práva přístupu a poslední parametr je příznak určující, že mapování nesmí být zrušeno v případě nedostatku paměti. Implementace této a podobných funkcí je závislá na

[4] Tyto struktury jsou alokovány v poli fixní velikosti určené při bootu systému. Pokud pole přeteče, jsou stránky odmapovány. Bylo by efektivnější struktury alokovat dynamicky.

architektuře, ale rozhraní je pro všechny architektury stejné. To umožňuje systém virtuální paměti rozdělit na část závislou na architektuře a část nezávislou na architektuře. O správu tabulek stránek se stará kód v souboru `pmap.c` a zbytek systému virtuální paměti pouze volá funkce tohoto souboru, pokud potřebuje nějakou stránku namapovat nebo odmapovat.

Při výpadku stránky systém najde `vm_map_entry` a objekt, kde k výpadku došlo. Pokud se požadovaná stránka nachází v objektu, natáhne se z jeho backing store a namapuje se. Pokud se stránka v objektu nenachází, hledá se dále v backing objektu tohoto objektu. Pokud je stránka nalezena v backing objektu, provede se kopie stránky (pokud se při výpadku stránky zapisovalo) a nová stránka se uloží do aktuálního objektu. Pokud v backing objektu stránka není, hledá se v backing objektu tohoto objektu a tak se postupuje dále až ke kořeni stromu objektů. Pokud stránka není ani tam, vytvoří se prázdná stránka a ta se uloží do nejhornějšího objektu, kde výpadek stránky vznikl.



Nové dva vm objekty vzniklé po provedení `fork`

Objekty se vytvářejí následovně:

- Proces dostane jeden objekt pro jeho alokovanou paměť (alokace pomocí `malloc` se vnitřně děje přes `mmap` s příznakem `MAP_ANONYMOUS`).
- Pokud proces udělá `fork`, vyrobí se dva nové objekty pro dva pokračující procesy. Oba objekty nemají na počátku žádné stránky a mají backing store nastavený na swapovou oblast. Původní objekt již nenáleží žádnému procesu, ale stane se z něj backing objekt pro oba nové objekty. Zde vidíme, že začne fungovat copy-on-write: při výpadku stránky se stránka nenalezne v objektu procesu, začne se vyhledávat v backing objektu, kde je nalezena, a pořídí se její kopie.
- Pokud proces namapuje soubor pro čtení nebo pro sdílený zápis, vytvoří se `map_entry`, která bude ukazovat na objekt příslušející souboru.
- Pokud proces namapuje soubor pro privátní zápis, vytvoří se nový objekt pro modifikované stránky souboru. Objekt má backing store swap a neobsahuje na počátku žádné stránky. Backing objekt pro tento nový objekt je nastaven na objekt příslušející namapovanému souboru. Pokud bude proces do stránek zapisovat, stránky budou kopírovány ze souboru do privátního objektu a v něm budou modifikovány.

  Datové struktury sice vypadají složitě, ale požadavky splňují dokonale.

FreeBSD má novou vlastnost — nazvanou IOOPT (povolení této schopnosti se provádí přidáním řádku „options ENABLE_VFS_IOOPT“ do konfiguračního souboru a překompilováním jádra). IOOPT spočívá v tom, že pokud proces čte velký soubor pomocí funkce `read`, není tento soubor kopírován do adresního prostoru procesu, ale stránky se

souborem jsou do adresního prostoru namapovány. Funguje to samozřejmě pouze, pokud je adresa čtení a offset v souboru na hranici stránky. Pokud proces do takových stránek zapíše, provede se copy-on-write. Copy-on-write se ovšem také musí provést, pokud do souboru zapíše jiný proces pomocí funkce `write`. IOOPT je experimentální vlastnost, není implicitně povolená a je možné, že bude obsahovat chyby.

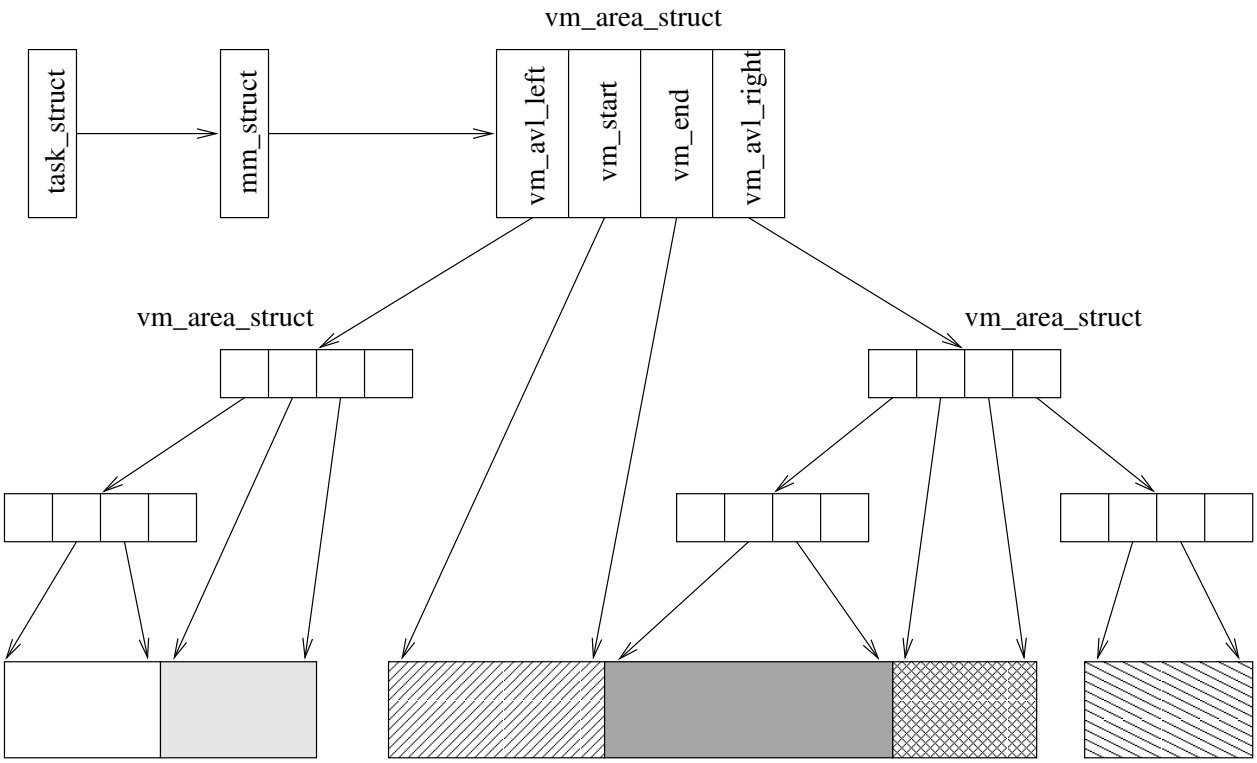
10.3. Struktury virtuální paměti na Linuxu

Linux má výrazně jednodušší správu virtuální paměti. Je vidět, že memory management na Linuxu vznikal spontánně, bez většího rozmyšlení. Původní Linux 0.01 měl jen copy-on-write po `fork`, neměl mapování ani sdílení stránek. V kódu jsou vidět poznámky Linuse „dnes jsem udělal load-on-demand“, „dnes jsem udělal sdílené stránky“ a podobné. Page cache přišla až v Linuxu 2.0.

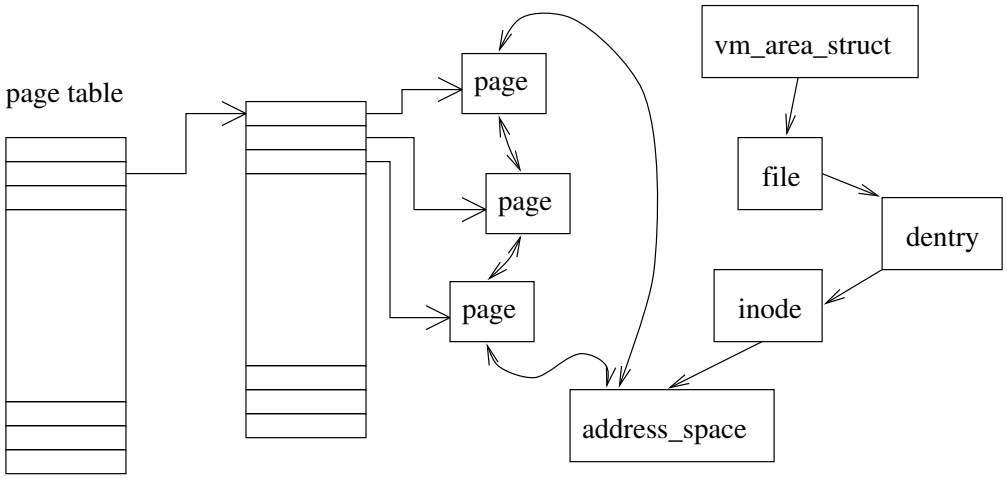
Pro každou fyzickou stránku existuje struktura `struct page`. Každá stránka má počítadlo použití — toto počítadlo určuje, z kolika míst někdo na stránku ukazuje. Funkce pro uvolnění stránky `free_pages` toto počítadlo zmenší o jedna a stránku uvolní pouze v případě, že počítadlo dosáhne nuly. Tím je zajištěno, že nebude uvolněna stránka, na kterou se někde nějaký kód odkazuje. Stránky na Linuxu mají pouze toto počítadlo mapování a nemají seznam všech mapování, jako je tomu u FreeBSD. Linux nemá žádné vm objekty. Na stránky ukazuje ukazatel z tabulky stránek procesu a/nebo jsou v hashové tabulce inody souboru, ke které náleží. Každý proces má `struct mm_struct`, která popisuje jeho adresní prostor (je to ekvivalent `struct vmspace` na FreeBSD). `mm_struct` obsahuje AVL-strom struktur `struct vm_area_struct`. Každá `vm_area_struct` odpovídá jednomu segmentu namapovanému pomocí `mmap` (je to ekvivalent `vm_map_entry` na FreeBSD). Struktury jsou uloženy ve vyváženém stromě, takže vyhledávání je rychlejší než vyhledávání v lineárním seznamu. `vm_area_struct` ukazuje rovnou na soubor, který je namapován — objekty zde neexistují.

Linux ukládá informace do samotných tabulek stránek. Pokud je stránka namapovaná, je možno zjistit `struct page` přímo z tabulky stránek po vhodném přepočítání fyzické adresy na offset do pole stránek. Pokud je stránka odswapovaná, do tabulky stránek se zapíše ukazatel do swap souboru, kde se stránka nachází. Linux nemá tak striktně jako FreeBSD oddělenou část závislou a nezávislou na architektuře. Celý kód správy paměti v Linux předpokládá, že poběží na systému s 3úrovňovými tabulkami stránek. Velikost těchto tabulek stránek a makra pro bitovou manipulaci s nimi jsou v hlavičkových souborech příslušejících každé architektuře. Pokud Linux běží na architektuře s 2úrovňovými tabulkami stránek (například IA32), jsou tato makra udělaná tak, že předstírají, že prostřední tabulka stránek má jen jednu položku. Tento přístup je sice jednodušší než na FreeBSD, ale má menší schopnosti — například na 64bitových architekturách není možno pokrýt celý 64bitový adresní prostor, neboť na to tabulky stránek tří úrovní nestačí. Na architekturách, které nemají vůbec tabulky stránek (jako například Sparc64), se přesto tabulky stránek musí vytvářet.

Při provedení `fork` Linux zkopíruje strom `vm_area_struct` a tabulky stránek. Anonymní alokované stránky nastaví jako read-only. U všech stránek zvětší jejich počítadlo použití, neboť na stránky se již odkazují dvě tabulky.



AVL strom popisující virtuální adresní prostor procesu



Tabulky stránek, stránky a struktura `address_space`

Při výpadku stránky Linux najde danou `vm_area_struct` a položku tabulky stránek. Pokud je položka tabulky stránek nulová, je buď alokována a vynulována nová stránka (pokud se jedná o anonymní alokovanou paměť), nebo je stránka natažena ze souboru (pokud se jedná o namapovaný soubor). Pokud položka tabulky stránek není nulová (ale má vynulovaný „present“ bit), předpokládá se, že ukazuje do swap souboru, a stránka je odtud natažena. Pokud dojde k pokusu o zápis do read-only stránky, Linux se podívá na počítadlo použití. Pokud je jedna, nastaví položku tabulky stránek na read-write a pokračuje. Pokud je více než jedna, provede kopii stránky a tuto nastaví jako read-write v tabulce stránek. Tak je zajištěna funkce copy-on-write při provádění `fork`.

Mapa swapu má pro každou stránku swap oblasti počítadlo použití. Pokud je nulové, stránka je volná. Pokud je jedna, ve stránce se nacházejí data. Pokud je větší než jedna, jedná se o odswapovanou stránku sdílenou po forku.

Po zapsání stránky do swap oblasti není stránka uvolněna, ale je umístěna do tzv. swap cache. Do swap cache se rovněž umisťují stránky ihned po natažení ze swap oblasti. Swap cache je hashovaná podobně jako stránky náležející souborům. Nacházejí se v ní stránky, které jsou jak ve swapu, tak v paměti. Do těchto stránek je zakázáno zapisovat — při pokusu o zápis se stránka ze swap-cache odstraní a její místo ve swapové oblasti je odalokováno. Stránky ve swap cachi je možno kdykoli uvolnit a není třeba je zapisovat do swapové oblasti. Swap cache zajišťuje, že pokud program bude procházet velké množství stránek a tyto stránky bude pouze číst, tak se stránky umístí do diskové swap oblasti i do swap cache a pak se budou z disku pouze číst, aniž by bylo nutno je zapisovat zpět.

Výše popsané principy umožňují splnit téměř všechny požadavky na virtuální paměť, které byly specifikovány — vyjma jediného — sdílená paměť. Sdílení stránek po `fork` a jejich copy-on-write funguje, ale sdílení stránek alokovaných pomocí SYSV SHM a jejich swapování zde není možno jednoduše implementovat. Verze Linuxu 2.2 a nižší to měly udělané značně komplikovaně — bylo vidět, že tam sdílená paměť byla uměle dodělávána. Linux 2.4 na sdílenou paměť pohlíží jako na filesystém (TMPFS), který nemá žádné fyzické blokové zařízení a jehož stránky jsou zapisovány do swapu. Používání sdílené paměti je implementováno jako mapování stránek z tohoto filesystému. Pokud uživatel chce, může si TMPFS i namountovat někam do adresářového stromu a mít pak filesystém, jehož obsah se ukládá do paměti a swapuje se do swapové oblasti. Zavedením TMPFS byl kód obsluhující sdílenou paměť výrazně zjednodušen.

Závěr: Linuxový memory management je podstatně jednodušší než na FreeBSD a obsahuje mnohem menší množství struktur. Nelze jednoznačně rozhodnout, zda je to lepší, nebo ne. Menší množství struktur znamená menší čas strávený jejich údržbou a tedy větší rychlost. Na druhou stranu absence seznamu mapování stránky může výrazně zpomalit prohledávání na uvolnitelné stránky (na FreeBSD se ke stránce ihned najde seznam všech mapování; je vidět, zda je stránka nepoužitá, a stránka se pak uvolní; na Linuxu se musí procházet všechny tabulky stránek a stránka se uvolní, až když je odmapovaná ve všech tabulkách). FreeBSD IOOPT je na Linuxu také naprosto neimplementovatelné, neboť jednoduché struktury Linuxu tak složité operace se stránkami neumožňují.

10.4. Základní algoritmy výměny stránek

Zde se stručně zmíním o základních algoritmech, které jsou popsány v literatuře. Tyto algoritmy samy o sobě nestačí, neboť neřeší problém sdílení stránek. Algoritmy používané ve skutečných operačních systémech jsou většinou kombinací těchto základních algoritmů.

- **LRU** — Least-recently-used. Vyrobí se fronta nacachovaných stránek. Když se přistupuje na stránku, která už v cachi je, nebo když se natáhne nová stránka z disku, umístí se na začátek fronty. Když dochází paměť, tak se stránky z konce fronty uvolňují z paměti a případně se zapisují na disk, pokud byly modifikovány. Tento algoritmus je možno implementovat na nenamapované stránky (data čtená pomocí syscallu `read`), ale není ho možno implementovat na stránky namapované v adresním

prostoru procesu, neboť procesor neumí při přístupu na stránku udělat operaci „přesun stránky na začátek fronty“[5]. Při přístupu na stránku procesor umí pouze nastavit v tabulce stránek jeden bit. Algoritmus LRU je velmi dobrý, většina implementací se mu snaží přiblížit, nicméně v jednom případě naprosto selže — při sekvenčním čtení souboru delšího, než je velikost paměti. Operační systém chovající se striktně podle LRU by v takovém případě měl odswapovat všechny procesy na disk a celou paměť použít jako cache na koncový úsek souboru. Takové chování je silně nežádoucí.

- **Hodinový algoritmus** — Stránky v paměti jsou v kruhovém seznamu. Existuje ukazatel na aktuální stránku (hodinová ručička). Při přístupu na stránku se nastaví bit, že na stránku bylo přistupováno. Při nedostatku paměti se podíváme na stránku, na kterou ukazuje ručička — pokud je bit přístupu smazaný, stránku uvolníme. Pokud je bit přístupu nastavený, smažeme jej a postoupíme na další stránku (pokud byly bity na všech stránkách nastavené, oběhneme jednou celý kruh a pak již nalezneme stránku, jejíž bit jsme dříve smazali). Tento algoritmus je možno použít i na namapované stránky, neboť procesor bit při přístupu nastavuje. Nevýhoda algoritmu je taková, že pokud nedochází k nedostatku paměti, algoritmus nic nedělá (kromě nastavování bitů), a až pak k nedostatku paměti dojde, vůbec se nerozpoznají stránky, na které bylo naposledy přistupováno před pěti sekundami nebo před deseti minutami. Hodinový algoritmus má stejně jako LRU nežádoucí chování při čtení dlouhého souboru.
- **Page aging** — Počítání „věku“ stránky. Stránka má kromě bitu přístupu ještě počítadlo age. Algoritmus funguje stejně jako hodinový algoritmus až na to, že navíc manipuluje s age. Při zavedení stránky je age nastaven na nějakou malou fixní hodnotu (může to být 1). Při zjištění nastaveného bitu přístupu je age zvětšen až do nějaké maximální meze, bit přístupu je smazán a postoupí se na další stránku. Při zjištění nenastaveného bitu přístupu je age snížen; pokud dosáhne 0, stránka je uvolněna. Pokud je větší než 0, postoupí se na další stránku. Tento algoritmus částečně řeší problém čtení dlouhého souboru — pokud je soubor větší než paměť, budou stránky souboru při dalším průchodu mít age 1, zatímco ostatní stránky, které byly v cachi dříve a na které se déle přistupovalo, budou mít age větší. Algoritmus bude tedy přednostně uvolňovat později načtené stránky souboru a ostatní stránky nějakou dobu v paměti nechá. Řešení to stále není ideální — všem stránkám se bude age snižovat, takže pokud budeme dlouhý soubor číst dostatečně dlouho, ostatním stránkám přece jen za čas klesne age na 0 a budou uvolněny. Dojde k tomu však mnohem později než v případě předchozích algoritmů.

10.5. Algoritmus výměny stránek na FreeBSD

Natahování stránek je zřejmé (systém prostě natáhne stránku, na které došlo k výpadku, případně ještě udělá read-ahead na několik dalších stránek). Nejsložitější částí systému virtuální paměti je algoritmus, který rozhoduje, jaká stránka se má uvolnit v pří-

[5] Šlo by to implementovat na architekturách se softwarovým zpracováváním TLB missů, jako například Sparc64, nicméně by to zpracování TLB missů zpomalovalo.

padě nedostatku paměti. FreeBSD používá čtyři fronty stránek: ACTIVE, INACTIVE, CACHE a FREE (množství stránek na těchto frontách je možno vidět ve výpisu příkazu `top`).

Fronta ACTIVE obsahuje stránky, na které se často a hodně přistupuje. Fronta INACTIVE obsahuje stránky, na které se málo přistupuje. Fronta CACHE obsahuje čisté nenamapované stránky, které je možno kdykoli uvolnit. (Název fronty „CACHE“ nemá nic společného s pojmem „cache“ používaným pro paměť obsahující data souborů z disku — do CACHE fronty se ukládají jak stránky obsahující cachované soubory, tak stránky alokované a swapované — stejně tak stránky obsahující cachované soubory se vyskytují nejen na frontě CACHE, ale i na ACTIVE a INACTIVE). Fronta FREE obsahuje volné stránky. Algoritmus se snaží držet určitý pomět mezi frontami ACTIVE a INACTIVE (typicky ACTIVE ¡ 2/3 celkové paměti). Dále se snaží, aby ve frontách CACHE+FREE bylo alespoň kolem 10MB paměti a aby ve frontě FREE bylo alespoň 500kB. Běžný kód v jádře může alokovat stránky z front CACHE a FREE (pokud je ve FREE více než 500kB, alokuje se z FREE, jinak se alokuje z CACHE); interrupty mohou alokovat pouze z fronty FREE. Oněch 500kB je paměť rezervovaná pro alokaci z interruptů (typicky se zde alokují buffery pro síťové packety).

Algoritmus má dvě nezávislé části: první část zajišťuje, aby ACTIVE fronta příliš nerostla — pokud je její velikost větší než nastavený limit, začnou se stránky z ACTIVE fronty přesouvat do INACTIVE fronty. K výběru stránky se používá algoritmus „Page aging“. Systém je možno přepnout, aby k výběru používal hodinový algoritmus (pokud sysctl `vm.pageout_algorithm != 0`). Druhá část algoritmu začne pracovat, pokud klesne množství paměti ve frontách CACHE a FREE. Tato část přesouvá stránky z fronty INACTIVE do fronty CACHE. Stránka je případně ještě zapsána do souboru nebo do swapu, pokud byla modifikovaná (t.j. má nastaven příznak `PG_DIRTY)`. Algoritmus výběru stránky z INACTIVE fronty je následující: vezme se stránka z konce fronty, pokud má nastaven příznak `PG_REFERENCED` (nebo pokud některé její mapování má nastaven příznak přístupu), tak se tento příznak zruší a stránka se umístí do ACTIVE fronty. Pokud stránka příznak `PG_REFERENCED` nemá (tzn. od doby jejího umístění do INACTIVE fronty ji žádný proces nečetl), tak je stránka buď zapsána (pokud má `PG_DIRTY`), nebo rovnou přesunuta do CACHE fronty. Experimentálně se zjistilo, že poněvadž zápis stránek je pomalý, je lepší uvolňovat čisté stránky než zapisovat špinavé — algoritmus byl modifikován tak, že špinavá stránka musí celou INACTIVE frontou projít dvakrát, než bude zapsána. Když je špinavá stránka nalezena poprvé, umístí se na začátek fronty a nastaví se jí příznak `PG_WINATCFLS`, který znamená, že stránka je v druhém průchodu. Pokud je na konci fronty nalezena stránka s `PG_WINATCFLS`, je zapsána a opět umístěna (už jako čistá stránka) na začátek INACTIVE fronty. Až po třetím průchodu INACTIVE fronty je stránka přesunuta do CACHE.

Pokud jsou nové stránky čteny nebo zapisovány pomocí `read` nebo `write`, jsou umístěny do fronty INACTIVE. Pokud jsou stránky čteny jako důsledek page-faultu, jsou umístěny do fronty ACTIVE.

Tento algoritmus splňuje většinu požadavků — pokud uživatel pouze čte nebo zapisuje soubor větší než paměť, budou stránky tohoto souboru umisťovány do INACTIVE fronty, ACTIVE fronty se to ani netkne, a proto tato operace nezpůsobí odswapovávání použitých stránek nebo uvolňování často používaných cachovaných stránek. Pokud bu-

deme číst veliký soubor, smaže nám to jen celou INACTIVE frontu, což je asi 1/3 paměti — na rozdíl například od algoritmu LRU, který by v takovém případě přemazal celou paměť. Page aging na ACTIVE frontě je také dobrá strategie, neboť to částečně zabraňuje přemazání celé ACTIVE fronty, pokud nějaký proces namapuje velké množství paměti (větší množství, než kolik je fyzické paměti) a pak na ni cyklicky přistupuje.

Ve starších verzích FreeBSD platilo, že stránky byly před přesunutím do INACTIVE fronty odmapovány. Byla snaha udělat jakousi napodobeninu LRU (jak víme, LRU nelze používat pro namapované stránky) — v ACTIVE frontě se stránky odmapovávaly a odstraňovaly pomocí page agingu a na INACTIVE frontě fungovalo LRU. Později se zjistilo, že odmapovávání a nové namapovávání stránek je náročné, proto se od odmapovávání upustilo a na INACTIVE frontě se nacházejí i namapované stránky.

FreeBSD má kromě klasických limitů na velikost dat a velikost celkové virtuální paměti procesu i limit na množství aktuálně namapovaných stránek. Pokud proces limit překročí, jsou mu stránky odmapovávány a přesouvány do INACTIVE fronty. Tento mechanismus umožňuje základní obranu proti vytrashování systému velkým procesem; nicméně není to ochrana absolutní — proces stále může vytrashovat ACTIVE frontu, ale nejde to jednoduchým mapováním velkého množství stránek.

FreeBSD má pokus o swapování celých procesů (je třeba povolit pomocí sysctl). Pokud proces dlouho neběžel nebo pokud se nepodařilo uvolnit dost stránek procházením front, dojde k tzv. swapoutu procesu. Při swapoutu jsou všechny stránky procesu přemístěny do INACTIVE fronty a tři stránky obsahující zásobník jádra jsou změněny na swapovatelné a také přesunuty do INACTIVE fronty. Při opětovném naswapování a rozběhnutí procesu je zásobník jádra natažen do paměti a vyjmut ze swapovacího systému, aby nebyl znovu odswapován. V dnešní době velikých pamětí již nemá žádnou cenu snažit se uvolnit tři stránky na proces (nehledě na to, že proces může zabírat mnohem více neswapovatelné paměti v handlech, strukturách file, mapování paměti a podobně) — tento kód na swapování zásobníku jádra zjevně pochází z velmi dávných dob, kdy bylo třeba stránkami velmi šetřit. Swapování celých procesů na FreeBSD nefunguje zdaleka tak čistě jako swapování procesů na VMS, neboť po naswapování procesu do paměti nejsou zavedeny stránky, které byly namapovány při odswapování procesu.

10.6. Algoritmus výměny stránek na Linuxu 2.2

V jádrech 2.2 a nižších se nacházel ne příliš kvalitní algoritmus. Algoritmus byl napsán spontánně, bez většího rozmýšlení, a byl laděn tak dlouho, dokud nefungoval rozumně. Algoritmus měl dvě části. První — nejdříve puštěná část — procházela ve stylu „hodinového algoritmu“ všechny stránky v systému. Pokud měla stránka nastaven bit přístupu `PG_referenced`[6], byl tento příznak smazán a stránka přeskočena. Pokud byl příznak přístupu vynulován a stránka byla namapovaná, byla stránka přeskočena. Pokud byl příznak přístupu vynulován a stránka nebyla namapovaná, byla stránka uvolněna (nebylo třeba stránku zapisovat, neboť v tomhle stavu se mohly nacházet pouze čisté

[6] nejedná se o příznak přístupu nastavovaný hardwarem — procházíme fyzické stránky a ne mapování

stránky).

Pokud se v první části algoritmu nepodařilo uvolnit určité množství stránek po určitém počtu kroků, spustila se druhá část. Tato část procházela cyklicky tabulky stránek všech procesů a podle principu „hodinového algoritmu“ odmapovávala stránky. Pokud byl nastaven hardwarový příznak přístupu, příznak byl snulován a stránka přeskočena. Pokud byl hardwarový příznak snulován, systém se podíval na hardwarový příznak modifikace. Pokud byl nastaven, stránka byla zapsána do swap oblasti (a umístěna do swap cache) nebo do filesystému. Pokud byl hardwarový příznak modifikace vynulován, stránka byla odmapována a zmenšilo se její počítadlo použití. Až byla takto nalezena a zrušena všechna mapování, stránka mohla již být uvolněna první částí algoritmu.

Algoritmus byl vyvinut postupným vývojem, není moc čistý, ale ve většině zátěží funguje rozumně. Algoritmus balancuje mezi použitím paměti pro alokované stránky a pro cache. Stránky filesystémové cache jsou uvolňovány dříve (k jejich uvolnění stačí jen první část algoritmu), zatímco stránky namapované ze souboru nebo alokované potřebují k uvolnění obě části algoritmu. Nemůže tedy dojít k velikému vyswapovávání procesů při používání cache. Samotnou cache však je možno vytrashovat při čtení velikého souboru. Další nevýhodou algoritmu je, že při určitém vzoru přístupu do cache nejsou stránky procesů nikdy odmapovávány ani odswapovávány — i kdyby tyto stránky zůstaly v paměti netknuté libovolně dlouho.

10.7. Algoritmus výměny stránek na Linuxu 2.4

Do Linuxu 2.4 byl napsán lepší algoritmus výměny stránek, který částečně vychází z FreeBSD. Algoritmus má fronty podobně jako u FreeBSD: ACTIVE a INACTIVE (kdysi měl i frontu CACHE, ale byla odstraněna — nejspíš kvůli jednoduchosti). Každá stránka má příznak přístupu `PG_referenced`. Jedna část algoritmu (funkce `refill_inactive`) přesouvá stránky z ACTIVE fronty do INACTIVE. Počet přesunutých stránek roste, pokud je ACTIVE fronta větší. Stránky jsou přesouvány podle hodinového algoritmu (kdysi se používal page aging, ale nemělo to dobrý výsledek). Druhá část algoritmu (funkce `shrink_caches`) odstraňuje stránky z konce INACTIVE fronty a v případě nutnosti je zapisuje na disk. Tato část nebere ohled na příznak `PG_referenced` — prostě odstraňuje stránky z konce fronty. Pokud stránka má mapování nebo nemůže být uvolněna z jiných důvodů, je přesunuta na začátek INACTIVE fronty. Pokud je při průchodu nalezeno větší množství namapovaných stránek, pustí se třetí část algoritmu (funkce `swap_out`), která prochází tabulky stránek procesů a podle hodinového algoritmu stránky odmapovává. Tato část je podobná jako v jádrech 2.2 (až na to, že neprovádí zápis stránky do swapu) a dokonce obsahuje kód pocházející ze starých jader.

Algoritmus trochu komplikuje fakt, že je dělený na jednotlivé zóny (připomínám, že na IA32 máme tři zóny: ISA DMA (velikost 16M), přímo mapovaná zóna (velikost 1G) a ostatní stránky nad 1G. Na 64bitových architekturách je možno namapovat celou paměť a stačí tam jedna zóna). Pro každou zónu existuje zvlášť počítadlo volných stránek, pokud u některé zóny množství volných stránek dosáhne kritické hranice, spustí se algoritmus jen pro tu danou zónu. Jednotlivé funkce algoritmu jako parametr dostávají zónu a přeskakují stránky, které v dané zóně neleží. Fronty ACTIVE a INACTIVE existují jen

jednou; na zóny děleny nejsou.

Pokud je stránka čtena pomocí funkce `read`, pokud je (v třetí části algoritmu) nalezeno mapování s příznakem přístupu nebo pokud je stránka namapována nějakému procesu, je zavolána funkce `mark_page_accessed`. Pokud má stránka vynulován příznak `PG_referenced`, je příznak nastaven. Pokud stránka má nastaven příznak `PG_referenced` a je v INACTIVE frontě, je příznak vynulován a stránka je umístěna do ACTIVE fronty.

Tento algoritmus splňuje většinu požadavků kladených na systém virtuální paměti. Namapované stránky jsou odmapovávány jen tehdy, když se jich moc nalézá v INACTIVE frontě. Nehrozí tedy, že by čtení dlouhého souboru způsobovalo odswapovávání většího množství stránek. Pokud uživatel sekvenčně čte soubor, který je delší než paměť, zasáhne tento soubor pouze INACTIVE frontu. Neboť každá stránka souboru je čtena jen jednou, funkce `mark_page_accessed` jí nastaví příznak `PG_referenced`, ale nepřesune ji do ACTIVE fronty. Algoritmus stránku z konce INACTIVE fronty odstraní, aniž by se jakkoli dotkl stránek v ACTIVE frontě. Pokud byla stránka přečtena více než jednou, dostane se do ACTIVE fronty a odtud již nebude tak snadno odstraněna. Namapované stránky se vždy dostávají do ACTIVE fronty — začnou sice v INACTIVE s nastaveným `PG_referenced`, ale při prvním pokusu o odstranění mapování je zjištěn příznak přístupu na mapování a stránka je přesunuta do ACTIVE fronty.

10.8. Chyby ve virtuální paměti

Elementární programátorské bugy se vyskytují ve všech druzích kódu. Většina bugů se dá snadno najít a odstranit, proto nepředstavují závažnější problém. Virtuální paměť je specifická tím, že se v ní vyskytují neodstranitelné bugy. Vývojáři o těchto chybách vědí, nicméně nedokáží je odstranit. K odstranění by bylo třeba zásadně přepsat nejen virtuální paměť, ale i filesystému a dokonce i síťování. V současných verzích systémů se proto vývojáři soustředí spíše na zmenšení pravděpodobnosti výskytu bugu než na jeho úplné odstranění.

Zásadním problémem virtuální paměti je tzv. low-memory deadlock. Princip bugu je následující: v systému došla volná paměť; je potřeba nějaké stránky uvolnit; stránky jsou špinavé, takže je potřeba je zapsat na disk; k uvolnění stránek je však potřeba další paměť. Systém zapisování stránek tedy čeká, než bude uvolněna nějaká paměť, systém správy paměti zase čeká, než budou nějaké stránky zapsány, aby je mohl uvolnit. Výsledek je ten, že operační systém vytuhne.

Low-memory deadlock se může vyskytovat v nejrůznějších případech:

- Swapování stránek do swapové partition — zde se low-memory deadlock nevyskytuje, neboť kód pro swapování je napsán tak, že žádnou paměť nealokuje. I když v systému není žádná volná stránka, je možno stránky zapisovat do swapu. Některé specifické ovladače blokových zařízení však paměť potřebují k tomu, aby mohly provést operaci zápisu. Na Linuxu se například nedoporučuje swapovat na softwarovou RAID partition. Na FreeBSD paměť pro požadavek alokuje i samotný ovladač IDE disku a vývojáři prostě doufají, že k nedostatku systémové paměti nedojde.
- Zapisování modifikovaných stránek do souboru — zde se low-memory deadlock vyskytuje. Filesystém totiž potřebuje k provedení zápisu do souboru nějakou paměť

alokovat (na to, aby našel fyzický blok, který dané části souboru náleží, potřebuje přečíst nějaké sektory z disku). Low-memory deadlock se nevyskytuje při zapisování dat pomocí syscallu `write` — syscall `write` se totiž může zablokovat a počkat, než bude paměť dostupná. Low-memory deadlock se vyskytuje při zapisování namapovaných stránek, neboť tam se až příliš pozdě z tabulky stránek zjistí, že stránka byla modifikovaná a je třeba ji zapsat.

- Swapování stránek do swapového souboru — zde se low-memory deadlock vyskytuje — situace je analogická předchozímu případu.
- Zapisování modifikovaných stránek přes NFS — zde se low-memory deadlock vyskytuje — k zapsání dat přes síť je třeba alokovat síťový buffer pro odchozí packet a je nutné alokovat buffer pro příchozí potvrzovací packet.
- Swapování stránek přes NFS — analogické předchozímu případu.
- Zapisování dat na loopback device — loopback device je speciální blokové zařízení, které je možno nastavit, aby obsahovalo data z nějakého souboru. Na loopback device je možno namountovat filesystém, který je ve skutečnosti souborem na jiném filesystému. Pokud bude systém zapisovat špinavé buffery nebo stránky na loopback device ve snaze se špinavých bufferů nebo stránek zbavit a uvolnit paměť, ve skutečnosti tím bude produkovat ještě větší množství špinavých stránek na hostitelském filesystému, který obsahuje soubor s daty pro loopback device. Tato snaha může vést až k totálnímu vyčerpání veškeré volné paměti a zatuhnutí na předchozím případu „zapisování modifikovaných stránek do souboru“.
- Zapisování dat nacházejících se nad hranicí přímo namapované paměti — tato data není možno zapsat rovnou, ale je k tomu potřeba „bounce buffer“ (jak bylo popsáno v kapitole o mapování stránek). Pokud se nenacházejí žádné volné stránky v přímo namapované zóně, dojde k nekonečnému čekání na bounce buffer a k vytuhnutí. Tento problém se vyskytoval v raných jádrech 2.4. V současných jádrech 2.4 se už nevyskytuje — ta mají pro bounce buffery předalokovanou část paměti.

Vzhledem k tomu, že dnes se v typických zátěžích systému používá většina paměti jako cache, k low-memory deadlocku nedochází, neboť čisté cachované stránky je možno vždy uvolnit. Nicméně low-memory deadlock se stále může vyskytnout například při zápisu velikého souboru nebo pokud nějaký program alokuje veškerou paměť a pak swapuje.

Linux 2.0 řešil low-memory deadlock na filesystému tak, že bufferová cache byla schopna znovupoužívat buffery. Pokud v systému nebyla žádná volná stránka a byl vznesen požadavek na čtení bufferu, tak se nějaký buffer uvolnil (v případě, že se žádný čistý buffer nevyskytoval, tak se nějaký špinavý buffer zapsal) a tento uvolněný buffer se okamžitě použil pro nová data. Žádná paměť se nevracela ani nealokovala ze systému správy paměti. Pokud filesystém k zápisu potřeboval jen buffery a nepotřeboval žádnou další paměť, nemohlo k low-memory deadlocku při zápisu souboru nemohlo. K low-memory deadlocku mohlo dojít, pokud v systému nebyly žádné buffery nebo pokud filesystém potřeboval více bufferů současně. Linux 2.2 a vyšší znovupoužívání bufferů nemá.

Nejjednodušší způsob řešení low-memory deadlocku je rezervovat nějaké stránky, které smí použít jen proces, který odswapovává nějaká data. Tohle řešení funguje ve většině případů, ale není ideální a k deadlocku může stále dojít. Má následující problémy:

- Musíme znát horní mez pro množství paměti potřebné k zapsání stránky. Například

na filesystému FAT32 je možné, že při zapsání bloku souboru bude třeba přečíst a projít celou FAT tabulku, a ta může mít prakticky neomezenou velikost (až 2^{32} položek).

- Co když proces uvolňující paměť uvnitř filesystému čeká na zámek, který drží jiný proces. Onen jiný proces nemůže alokovat rezervované stránky a čeká, než bude nějaká paměť uvolněna.
- Pokud více procesů současně uvolňuje paměť, může i tato rezervovaná paměť dojít. Tento problém se vyskytuje na Linuxu, kde jakýkoli proces nemající paměť může z `alloc_pages` zavolat `try_to_free_pages`, což spustí výše uvedený uvolňovací algoritmus. Na FreeBSD uvolňování paměti provádí jen jeden kernel thread, proto se zde tento problém nevyskytuje.
- V případě, že jsou stránky zapisovány přes síť, je potřeba nejen odeslat data, ale i přijmout potvrzení od serveru. Alokace paměti pro toto potvrzení je velmi problematická — v době alokace packetového bufferu nevíme, jaký packet do něj bude uložen. Není tedy možno rozhodnout, zda buffer alokovat z rezervované zóny, nebo ne. Proti deadlocku při zapisování dat přes síť nám rezervování paměti nepomůže.
- Když proces alokuje z rezervované paměti a dělá zápis stránek, může tyto stránky zapisovat na loopback device — zápis tedy způsobí vyrobení spousty nových špinavých stránek a vyčerpání rezervované zóny.

Jediné, co se v současných systémech používá, je rezervace paměti pro swapující proces, ale i to má výše uvedené nevýhody. Nejlépe na tom byl Linux 2.0, neboť ten měl znovupoužívání bufferů, které low-memory deadlocku na lokálním filesystému zabraňovalo.

Problém loopback device se řeší tak, že limit pro špinavé buffery na loopback device je nižší než na ostatních zařízeních. Buffery pro loopback se tedy začnou zapisovat dříve a většinou nezaplní celou paměť. Toto řešení opět není stoprocentní a k vytuhnutí systému při použití loopback device občas opravdu dochází. Téměř určitě dojde k vytuhnutí, pokud přiřadíme loopback device soubor, který se nachází na filesystému na jiném loopback device.

Poučení z této kapitoly zní:

- Nemapovat veliké soubory pro zápis (možná to rezervace paměti vyřeší, ale 100% jistota to není).
- Už vůbec to nedělat, pokud jsou ty soubory na NFS (tady nám opravdu nepomůže vůbec nic a systém vytuhne).
- Neswapovat do lokálního souboru a už vůbec ne přes NFS. Swapovat na partition.
- Nepoužívat loopback device pro zápis. (V raném Linuxu 2.4 to skutečně často zatuhlo; v novějších jádrech byla pouze snížena pravděpodobnost vytuhnutí, ale problém vyřešen nebyl.)

10.9. Měření rychlosti filesystému a stránkové cache

Provedl jsem měření rychlosti filesystémových operací — čtení, zápis a otevírání souboru. V případě čtení byl opakovaně čten soubor dané velikosti pomocí jednoho syscallu `read`. V případě zápisu byl soubor dané velikosti opakovaně zapisován pomocí syscallu

`write`. Při měření rychlosti otevírání souboru byl opakovaně otevírán soubor, jehož adresářovou cestu tvoří daný počet položek.

velikost souboru	Linux 2.4.20	FreeBSD 4.7	FreeBSD 4.7+IOOPT
0	0.0024ms	0.0076ms	
16B	0.0044ms	0.0150ms	
64B	0.0052ms	0.0156ms	
256B	0.0089ms	0.0196ms	
1kB	0.0237ms	0.0259ms	
4kB	0.0824ms	0.0576ms	0.0271ms
16kB	0.3677ms	0.2290ms	0.0673ms
64kB	1.6985ms	0.9071ms	0.2124ms
256kB	6.7378ms	3.6088ms	0.9375ms
1MB	27.0778ms	18.0212ms	4.0668ms

Čtení souboru

Při čtení souboru je vidět, že FreeBSD pomaleji čte malé bloky dat. Je to dáno tím, že správa paměti na FreeBSD je složitější, a tak se spotřebuje více času na hledání inody a stránky. Delší soubory čte FreeBSD 1.5krát rychleji, neboť používá pro kopírování dat instrukce koprocesoru, které jsou na Pentiu rychlejší. Dá se předpokládat, že na jiných procesorech toto zrychlení nenastane. Za povšimnutí rovněž stojí, že bloky dat o velikosti 64kB a 256kB jsou na FreeBSD čteny více než 1.5krát rychleji — dá se předpokládat, že je to způsobeno barvením stránek. Test byl prováděn na počítači s 512kB L2 cache a při použití barvení bude 256kB dat souboru a 256kB adresního prostoru procesu, kam se data kopírují, optimálně umístěno, aby se co nejlépe vešlo do cache. Pokud je ve FreeBSD zapnut `VFS_IOOPT`, dojde ke značnému zrychlení, neboť se místo kopírování dat provádí pouze manipulace s tabulkami stránek.

Test zápisu není tak jednoduchý jako test čtení — při čtení se testuje pouze stránková cache a neprovádí se žádné operace s filesystémem, naproti tomu při zápisu se provádí alokokace bloků na disku — změny příslušející těmto alokacím se nezapisují ihned na disk, ale ukládají se do bufferové cache. Z testu vidíme jednoznačnou převahu Linuxu, neboť je jeho kód jednodušší. FreeBSD navíc nemá prealokaci bloků na disku. Je také vidět, že pro malé soubory je výhodnější menší velikost bloku (neboť blok za koncem souboru se musí nulovat, což zabere čas) a pro velké soubory je lepší větší velikost bloku, neboť je méně práce s alokací bloků a udržování informací o bloku. Test s velikostmi souboru 256kB a 1MB nebylo možno provést na FreeBSD, neboť FreeBSD začne tak veliké soubory ukládat na disk, což by výsledky testu značně zkreslilo.

Otevření souboru je test na rychlost cache pro vyhledávání v adresářích. Je vidět, že Linux je jednoznačně rychlejší díky své jednoduchosti. Na Linuxu se při otevírání souboru pouze prochází strom struktur dentry. FreeBSD má komplikovanější způsob práce s vnodami a s adresářovou cachí, proto je na něm otevření souboru pomalejší.

velikost souboru	Linux 2.4.20 (blok 1kB)	Linux 2.4.20 (blok 4kB)	FreeBSD 4.7 (f 512B/b 2kB)	FreeBSD 4.7 (f 2kB/b 16kB)
0	0.0069ms	0.0075ms	0.0290ms	0.0329ms
16B	0.0727ms	0.1335ms	0.1510ms	0.2422ms
64B	0.0747ms	0.1296ms	0.1517ms	0.2430ms
256B	0.0726ms	0.1294ms	0.1562ms	0.2464ms
1kB	0.0746ms	0.1295ms	0.1700ms	0.2567ms
4kB	0.1801ms	0.1301ms	0.2531ms	0.3213ms
16kB	0.6025ms	0.4146ms	0.8451ms	0.6661ms
64kB	2.2771ms	1.6361ms	3.5420ms	2.3681ms
256kB	8.9338ms	6.5052ms		
1MB	36.2681ms	25.7899ms		

Zápis do souboru

velikost cesty	Linux 2.4.20	FreeBSD 4.7
1	0.0152ms	0.0251ms
2	0.0171ms	0.0345ms
3	0.0193ms	0.0426ms
4	0.0216ms	0.0503ms
5	0.0238ms	0.0595ms
6	0.0259ms	0.0667ms
7	0.0279ms	0.0758ms
8	0.0299ms	0.0833ms
9	0.0320ms	0.0911ms
10	0.0339ms	0.0988ms

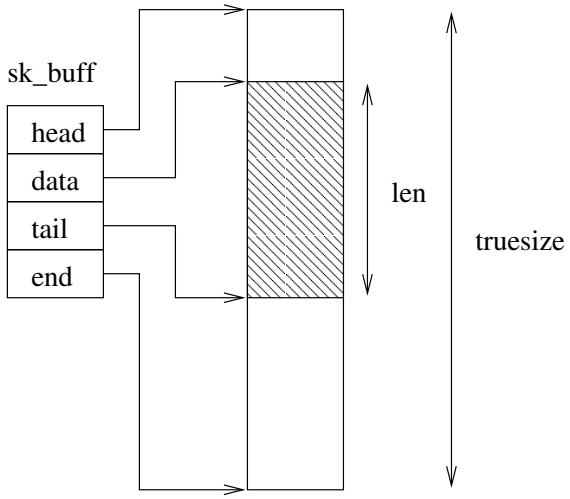
Otevření souboru

11. Síť

V této kapitole se budeme zabývat implementací síťového stacku. Je třeba, aby operační systém podporoval více protokolů. Mezi ovladači těchto protokolů se předávají packety, přičemž každý protokol přidá nebo odebere svoji hlavičku. Packet může putovat mezi rozličnými ovladači protokolů — například Ethernet—IP—TCP, Ethernet—IP—UDP, PPP—IP—TCP, Ethernet—IPX a jiné. Je třeba najít obecný formát, v jakém se packety mezi protokoly předávají.

11.1. Socket buffery na Linuxu

Linux k předávání packetů používá socket buffery. Socket buffer má hlavu a datovou oblast. Hlava je popsána strukturou `struct sk_buff`. Hlava obsahuje pointery `head` a `end`, které ukazují na začátek a konec alokované datové oblasti. Dále obsahuje pointery `data` a `tail`, které ukazují do oblasti určené `head` a `end` na data aktuální protokolové vrstvy. Socket buffer se alokuje pomocí funkce `skb_alloc`. V souboru `skbuff.h` je spousta dalších inlinovaných funkcí pro manipulaci s pointery nebo kopírování bufferu.



Socket buffer na Linuxu

Když je socket buffer vytvářen, rezervuje se na začátku dostatečné množství bytů pro hlavičky všech protokolů, pak se nakopírují vlastní data (na pointer `data`) a na konec se ukáže pointerem `tail`. Pak si jednotlivé vrstvy (např. TCP, IP, Ethernet) postupně posouvají `data` k nižším adresám a vkládají tam svoje hlavičky. Nakonec se socket buffer dostane k příslušnému ovladači síťové karty a ten odešle všechny byty mezi pointery `data` a `tail`. Při posouvání `data` směrem dolů je třeba zajistit, aby tento pointer nikdy nebyl menší než pointer `head`. Pokud k tomu dojde, jádro spadne na `panic` (může k tomu dojít jen tehdy, pokud někdo napsal nekorektní ovladač protokolu, který alokuje příliš mnoho hlaviček).

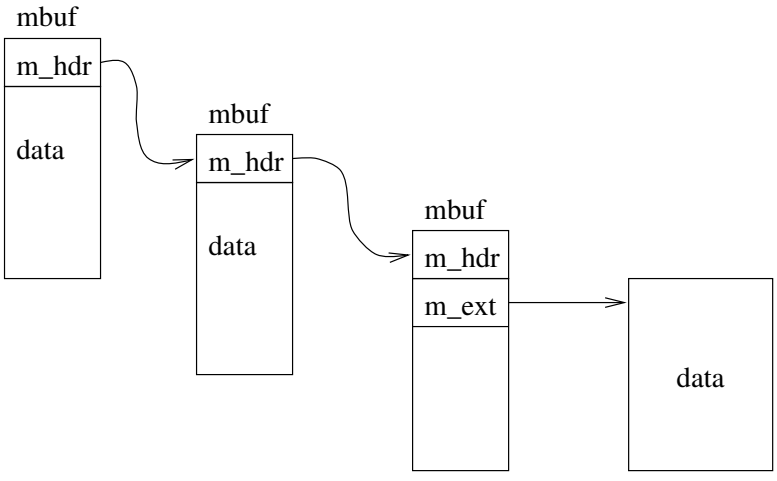
Pokud Linux přijímá packet, alokuje ovladač karty socket buffer o dostatečné velikosti, umístí do něj celý packet a nastaví `data` na začátek packetu a `tail` na konec packetu.

Poté je packet předán síťové vrstvě pomocí funkce `netif_rx`. Tato funkce rozhodne, ke kterému protokolu packet patří. Příslušný protokol si přečte svoji hlavičku, posune `data` za konec hlavičky a předá packet dalšímu protokolu. Až packet projde všemi protokoly, zbydou mezi pointery `data` a `tail` jen samotná data, která se předají uživatelskému programu.

Socket buffery je možno klonovat. Klonování je potřeba tehdy, pokud je nutno jedna data přijatá ze síťové vrstvy předat dvěma nebo více odlišným protokolovým stackům. Typický případ, kdy se klonování používá je prohlížení sítě pomocí programu `tcpdump` — v takovém případě je třeba příchozí packet předat protokolové vrstvě, které náleží, k dalšímu zpracování a RAW socketu, pomocí kterého `tcpdump` sleduje tok na síti. Při klonování se vytvoří několik struktur `struct sk_buff`, které ukazují na tatáž data. Za pointerem `end` je umístěna struktura `struct skbuf_shared_info` (je možno se na ni dostat pomocí makra `skb_shinfo`), která obsahuje sdílené informace. Nejdůležitější je počítadlo, kolik `sk_buf` se na tuto oblast odkazuje. Při smazání posledního klonu toto počítadlo dosáhne nuly a datová oblast je uvolněna.

11.2. Mbuf na FreeBSD

FreeBSD používá k popisu packetu struktury `struct mbuf`. Jeden packet je popsán řetězcem těchto struktur. Každá `struct mbuf` má pointer `m_next` na další část packetu. Pokud je tento pointer `NULL`, jedná se o poslední kus packetu. První mbuf má navíc strukturu `struct pkthdr`, která obsahuje obecné informace o packetu. Data se mohou nacházet buď rovnou ve struktuře mbuf, nebo v externí oblasti definované strukturou `struct m_ext`, která je součástí mbuf. `m_ext` obsahuje pointer na začátek dat, délku a funkce pro uvolnění a klonování. Každý mbuf má pevnou velikost 256 bytů (je možno nastavit v konfiguračním souboru při kompilaci jádra). Pokud se do něj data vejdou, uloží se tam za konec struktur, jinak se alokuje externí oblast. S mbufy je možno manipulovat pomocí maker v souboru `mbuf.h`.



Packet popsaný pomocí tří zřetězených struktur mbuf

Když je vytvářen mbuf s packetem k odeslání, vyrobí se mbuf obsahující pouze hla-

vičky protokolů IP a TCP. Tyto hlavičky jsou vyplněny a k tomuto mbufu je připojen nový mbuf obsahující ukazatele na data do fronty příslušného socketu. Pokud je dat málo, jsou nakopírována rovnou do původního mbufu za hlavičky.

Při přijetí packetu síťový ovladač sám alokuje a vyplní mbuf. Pokud je packet dlouhý, musí mbuf obsahovat pointer na externí data. Ovladač může specifikovat funkce pro uvolnění nebo zvětšení počítadla referencí pro tuto externí oblast.

Mbufy je možno klonovat podobně jako na Linuxu — `struct mbuf` se prostě zkopírují, a pokud mbuf obsahuje nějaká externí data, zavolá se na tato data speciální funkce, která zvětší počítadlo referencí.

11.3. Zero-copy TCP

Síťové servery zpravidla posílají veliké množství dat ze souborů do sítě. Velkého zrychlení docílíme, pokud budeme data posílat rovnou z page cache do síťové karty pomocí DMA bez jakéhokoli kopírování dat do síťových bufferů. Potřebujeme k tomu síťovou kartu, která umí tzv. scatter-gather DMA. Při scatter-gather DMA jsou ovladači síťové karty předány ukazatele na začátky a délky jednotlivých fragmentů, které tvoří packet. Síťová karta sama přečte z paměti jednotlivé fragmenty, poskládá packet a pošle ho na síť. Většina moderních PCI síťových karet scatter-gather DMA umí[1].

Program může pro zero-copy TCP použít syscall `sendfile`. Syscall pošle daný soubor na daný socket. Syntax syscallu se liší na Linuxu a na FreeBSD — na FreeBSD je možno pomocí syscallu navíc ještě připojit nějaká data před nebo za soubor.

Zero-copy TCP existovalo dlouho pouze na FreeBSD — zero-copy se velmi snadno implementuje pomocí řetězu mbufů (vyrobíme jeden mbuf obsahující TCP/IP hlavičky a druhý mbuf obsahující ukazatel na externí data do page-cache a tyto dva mbufy zřetězíme), ale nelze implementovat pomocí socket bufferů obsahujících data packetu v jedné lineární oblasti na Linuxu. Posledních několik verzí Linuxu 2.4 má také zero-copy — seznam externích fragmentů se ukládá na konec datové oblasti socket bufferu do struktury `struct skbuf_shared_info`. Poněvadž se na Linuxu jedná o novou vlastnost, nepodporují ji ještě všechny ovladače síťových karet. I když samotná karta scatter-gather DMA umí, není jisté, že někdo přepsal ovladač, aby uměl tyto fragmenty do scatter-gather seznamu karty ukládat.

Zero-copy s sebou přináší další problém — a to je počítání kontrolních součtů packetů. Každý TCP a UDP packet má mít kontrolní součet datové oblasti. Dokud zero-copy neexistovalo, součet se počítal při kopírování dat do fronty na socketu. Počítání součtu při kopírování dat nezpomaluje téměř vůbec. Aby data mohla být při zero-copy posílána rovnou z paměti na kartu a aby nemusela procházet pomalou sběrnicí mezi pamětí a procesorem, musí síťová karta být schopna spočítat kontrolní součet. Většina moderních síťových karet kontrolní součty počítat umí. Některé mají v sobě napevno zadrátované

[1] Zde musím varovat před velmi rozšířenými kartami Realtek RTL8139 — tyto karty nejenže neumí sg-DMA, ale navíc vyžadují zcela nevhodné zarovnání packetů pro DMA, proto se u nich celý packet musí kopírovat na správné místo, než může být odeslán. Ve vytížených serverech je použití takové karty zcela nevhodné.

počítání kontrolních součtů pro IP, některé umí počítat součty libovolné oblasti, takže jsou použitelné i pro IPv6 nebo jiné protokoly. FreeBSD i novější verze Linuxu 2.4 umějí počítání kontrolních součtů na kartě využít. Před nákupem karty do serveru je však třeba si ověřit, že karta i ovladač v daném systému počítání kontrolních součtů podporují — jinak přijde celé zero-copy TCP na nic a data se stejně budou muset kopírovat do procesoru pro spočítání součtu.

V experimentálním FreeBSD 5 bylo zavedeno zero-copy i při syscallu `write`[2](je nutno to povolit volbou `ZERO_COPY_SOCKETS` při kompilaci jádra). Pokud program zavolá `write`, zapisovaná stránka je označena v tabulce stránek jako read-only a vyrobí se mbuf, jehož externí data ukazují rovnou na tuto stránku. Takový mbuf se dále předá TCP/IP stacku, který před něj přidá mbuf s hlavičkami a odešle packet na síťovou kartu. Karta pak rovnou pomocí DMA čte data ze stránky náležící uživatelskému adresnímu prostoru. Pokud program mezitím do stránky zapíše, vyrobí se copy-on-write kopie stránky a původní stránka zůstane netknutá. Zero-copy se provádí pouze, pokud množství zapsaných dat je větší než stránka. Pokud je množství zapsaných dat malé, je rychlejší data zkopírovat než provádět manipulaci s tabulkami stránek. Zero-copy při `write` může v určitých situacích přinést zrychlení síťové komunikace, nicméně ve většině případů k tomu nedojde — je dost práce s manipulací s tabulkami stránek a navíc, aby to správně rychle fungovalo, program nesmí do oblasti paměti, kterou předal syscallu `write`, zapsat, dokud data nebudou odeslána a potvrzena. Většina programátorů ihned po zápisu bufferu pomocí `write` tento buffer uvolní nebo použijí na něco jiného, což způsobí kopírování stránek při pokusu o zápis do read-only stránky a celou věc to ještě více zpomalí.

[2] Zero-copy při `write` funguje již velmi dlouho na operačním systému NetBSD.

12. Rozšíření rozhraní mezi procesy a jádrem

Systémy Linux i FreeBSD podporují standardní unixové rozhraní specifikované normou POSIX. Toto rozhraní je velmi staré a v poslední době se začínají projevovat některé jeho nedostatky. Proto bylo třeba toto rozhraní rozšířit.

12.1. Čekání na více událostí

Při psaní síťového serveru vyvstává otázka, jaký zvolit model paralelního zpracování požadavků. Na klasickém Unixu server fungoval tak, že proces při přijetí každého požadavku udělal `fork` a nechal dětský proces zpracovávat požadavek, zatímco rodičovský proces čekal na další požadavek. Tento model je značně neefektivní, neboť `fork` je náročná operace a zpracovávání požadavků značně zpomaluje. Pokud server zpracovává paralelně veliké množství požadavků (např. ftp server může udržovat až několik tisíc spojení), zabírají informace spojené s procesy veliké množství paměti. Další možností je použití threadů místo procesů. Toto řešení se skutečně používá, nicméně stále je zde problémem časté přepínání threadů, velká zátěž na scheduler a množství paměti, které thready konzumují (každý thread potřebuje zásobník v jádře a v userspace a položku v tabulce procesů).

Pokud nechceme, aby každý požadavek zpracovával zvláštní proces nebo thread, musíme mít způsob, jak jeden thread bude moci čekat na více událostí současně. V Unixu se k tomu používá funkce `int select(int n, fd_set *rd, fd_set *wr, fd_set *except, struct timeval *timeout)`. `n` je číslo nejvyššího nastaveného handlu, `rd`, `wr` a `except` jsou ukazatele na bitová pole specifikující, které handly nás zajímají. `rd` jsou handly pro čtení, `wr` jsou handly pro zápis, `except` jsou handly, na která přišla urgentní out-of-band data. Funkce `select` zablokuje proces, dokud z některého handlu z množiny `rd` nelze číst nebo dokud na některý handle z množiny `wr` nejde zapisovat, dokud na některý handle z množiny `except` nedošla out-of-band data nebo dokud nevyprší `timeout`. Problémem funkce `select` je, že bitová pole mají omezenou délku — maximálně 1024 handlů. Funkci nelze použít k čekání na handly s vyššími čísly[1]. Proto byla v Unixu zavedena novější funkce `int poll(struct pollfd *fds, unsigned long nfds, int timeout)`. Handly, na které má čekat, nejsou uloženy v bitovém poli, ale v poli struktur `struct pollfd`. V každé `pollfd` se nachází číslo handlu, bitová maska událostí, na něž se má čekat, a bitová maska, do níž jádro vrátí události, které nastaly.

Obě funkce `select` i `poll` jsou pro čekání na větší množství událostí zcela nepoužitelné — důvodem je jejich časová složitost, která je úměrná počtu událostí, na které se čeká. Představíme-li si například ftp server s 5000 paralelními spojeními, je po vyřízení každého požadavku třeba zavolat funkci `select` nebo `poll`, která bude procházet 5000 handlů a kontrolovat, na kterém nějaká data jsou a na kterém ne. Procházení takového množství dat je pomalé, a proto je třeba zavést jiné rozhraní, ve kterém uživatelský pro-

[1] V novějších systémech je tento problém odstraněn a jádro je již schopno pracovat s bitovými poli libovolné délky (délka se určí z parametru `n`), nicméně makra v hlavičkových souborech na tuto změnu nejsou vždy připravena.

gram událost, na niž čeká, jednou zaregistruje a je upozorněn, až událost nastane, aniž by musel po vyřízení každého požadavku znovu registrovat všechny události, na které čeká.

12.2. Realtimové signály na Linuxu

Realtimové signály jsou standardizovány v normě POSIX. Původně nebyly určeny pro čekání na události. Jeden proces může jinému procesu poslat realtimový signál pomocí syscallu `rt_sigqueueinfo`. K signálu je možno přidat nějakou informaci ve struktuře `siginfo_t`. Cílový proces tuto informaci dostane jako parametr handleru signálu. Realtimový signál má tu vlastnost, že cílovému procesu dojde přesně tolikrát, kolikrát byl odeslán, pokaždé s jinou `siginfo_t`, která byla předána syscallu `rt_sigqueueinfo` (na rozdíl od obyčejných signálů, které dojdou jen jednou, pokud byly odeslány vícekrát dříve, než byl v cílovém procesu zavolán handler).

Realtimové signály je možno použít i pro čekání na události. Je známá věc, že pokud pomocí syscallu `fcntl(h, F_SETFL, FASYNC)` nastavíme na handlu příznak `FASYNC`, proces dostane signál `SIGIO`, pokud z handlu je možno číst nebo do něj zapisovat. Pokud navíc pomocí `fcntl(h, F_SETSIG, signal)` nastavíme číslo signálu jako realtimový signál, proces dostane realtimový signál, až bude možno z handlu číst nebo do něj zapisovat. Ve struktuře `sigingo_t`, kterou handler signálu dostane, se objeví číslo handlu, který signál způsobil.

Realtimové signály vypadají krásně, nicméně reálné použití už tak pěkné není. Obyčejné signály se posílají pouhým nastavením bitu v masce signálů, a proto k poslání signálu není potřeba žádná paměť. Realtimové signály potřebují paměť na frontu signálů (neboť signál musí dojít přesně tolikrát, kolikrát byl odeslán) a struktury `siginfo_t`. Aby paměť jádra nebyla zaplněna těmito strukturami, pokud si nějaký proces signály nevyzvedává, existuje v systému limit na maximální počet nevyřízených signálů držených v paměti. Existuje však mnohem horší problém — realtimový signál je často třeba poslat ze softwarového přerušení sítě (např. pokud došla nějaká data na socket). V softwarovém přerušení neexistuje kontext procesu, a proto se není možno zablokovat. Pokud se není možno zablokovat, není možno vždy alokovat paměť (neboť není možno čekat na swapper, než nějaké stránky odswapuje). Je možno alokovat paměť pouze s příznakem `GFP_ATOMIC` a taková alokace může kdykoli selhat. Pokud alokace selže, procesu není možno poslat realtimový signál, místo toho se mu pošle `SIGIO`, k jehož poslání žádná paměť třeba není. Pokud jeden proces pošle druhému procesu realtimový signál pomocí `rt_sigqueueinfo`, může to také kdykoli selhat — buď proto, že se přeplní systémový limit, nebo proto, že selže `GFP_ATOMIC` alokace (zde jsme sice v kontextu procesu, a proto by bylo možno alokovat pomocí `GFP_KERNEL` a čekat, než bude volná paměť, ale nedělá se to, protože poslání signálu je obsluhováno jednou rutinou, která může být volána jak z kontextu procesu, tak z kontextu přerušení). Pokud alokace při `rt_sigqueueinfo` selže, není posláno nic, návratová hodnota značí chybu a `errno` je nastaveno na `EAGAIN`.

Závěr: Pro čekání na události je použití realtimových signálů dost krkolomné — signál může být kdykoli ztracen — jediné rozumné použití je odchytit signál `SIGIO` a v jeho handleru pomocí `select` nebo `poll` zkontrolovat všechny handly, aby bylo možno vyřídit i události, od nichž byl realtimový signál ztracen.

12.3. kqueue na FreeBSD

Vývojáři FreeBSD neimplementovali realtimové signály, ale navrhli a implementovali vlastní rozhraní nazvané kqueue. kqueue je specifické pro FreeBSD a není standardizováno (na rozdíl od realtimových signálů). kqueue je implementováno na FreeBSD 4 i 5.

Syscall `int kqueue(void)` vyrobí frontu a vrátí handle této fronty. Syscall `int kevent(int kq, const struct kevent *changelist, int nchanges, struct kevent *eventlist, int nevents, const struct timespec *timeout)` umožňuje do fronty přidávat nebo z ní ubírat události, na které je třeba čekat, nebo vybírat seznam událostí, které nastaly. `kq` je handle fronty vrácený syscallem `kqueue`. `changelist` je seznam událostí, na které se má začít nebo přestat čekat (zda se má začít, nebo přestat čekat, je rozhodnuto příznakem ve struktuře `struct kevent`). `nchanges` je velikost tohoto seznamu, pokud je nulová, žádné události se přidávat ani ubírat nebudou. `eventlist` je místo v paměti o velikosti `nevents`, kam jádro uloží seznam událostí, které byly (při současném volání nebo někdy dříve) registrovány a které nastaly. Pokud je `nevents` nula, žádné události sem uloženy nebudou. Příznakem v `struct kevent` je možno určit, zda bude událost automaticky odebrána, pokud nastala a byla vrácena v `eventlist`, nebo se na ni bude čekat dál a bude moci být vrácena, až znovu nastane.

Události, na které je možno čekat, jsou mnoha druhů — kromě standardní možnosti číst nebo zapisovat do handlu je tu i možnost čekat na dokončení asynchronního IO (viz níže), změnu libovolného souboru nebo adresáře jiným procesem, ukončení, forknutí nebo execnutí jiného procesu nebo přijetí signálu.

Každá kqueue je implementována jako fronta událostí `knote`, které nastaly. Pokud proces registruje událost, vyrobí se struktura `knote`, která obsahuje pointer na kqueue, ke které náleží, a tato struktura se přiřadí do seznamu na příslušném handlu, procesu, souboru či jiné struktuře, na jejíž změnu stavu se čeká. Až událost nastane, jsou struktury `knote` ze seznamu zařazeny do odpovídajících kqueue, odkud si je uživatelské procesy mohou vybrat. Nedochází k problému, jaký má Linux s realtimovými signály — když událost nastane, tak je pouze struktura vybrána ze seznamu a uložena do fronty, k tomu není potřeba žádná paměť, takže nemůže nastat problém s nedostatkem paměti. Struktura `knote` je alokována při volání `kevent` v kontextu procesu, a tam je možno alokovat paměť vždy.

Závěr: kqueue je velmi dobrá implementace, která bohužel není standardizována. kqueue nemá problémy s přeplněním fronty a ztrácením událostí. Další velikou výhodou kqueue je možnost čekat nejen na čtení nebo zápis do handlu, ale i na množství dalších událostí. kqueue je možno snadno rozšířit o nové události.

12.4. epoll na Linuxu 2.5

Experimentální Linux 2.5 přichází s novým rozhraním `epoll`. Aplikační program vytvoří pomocí syscallu `epoll_create` handle, který bude dále používat k čekání na události (je to ekvivalent syscallu `kqueue` na FreeBSD). Události je možno registrovat pomocí `int epoll_ctl(int efpd, int op, int fd, unsigned events)`. První para-

metr je handle, který vrátil syscall `epoll_create`, druhý parametr je operace `EP_CTL_ADD`, `EP_CTL_DEL` nebo `EP_CTL_MOD` pro přidání, odebrání nebo modifikování události. Třetí parametr je handle, na kterém má být událost sledována, a čtvrtý parametr je typ události (používají se stejné hodnoty jako u syscallu `poll`). Události je možno vybírat pomocí syscallu `int epoll_wait(int epfd, struct pollfd *events, int maxevents, int timeout)`. První parametr je handle z `epoll_create`, druhý parametr je pole, do kterého budou události uloženy, třetí parametr je velikost pole a čtvrtý parametr je čas, po který bude funkce čekat, pokud žádné události nenastaly.

Závěr: Rozhraní `epoll` je podobné `kqueue` z FreeBSD, nenabízí však takové možnosti — pomocí `epoll` je možno čekat pouze na čtení nebo zápis do handlu a nikoli na dokončení asynchronního IO, ukončení procesu, signál nebo změnu adresáře (pro čekání na změnu adresáře je možno použít syscall `fcntl` s parametrem `F_NOTIFY`). Navzdory tomu je `epoll` výrazně lepší než realtimové signály na předchozích verzích Linuxu.

12.5. Asynchronní IO

Při implementaci jednothreadového serveru vznikne další problém — pokud proces čte nějaký soubor a zablokuje se při čekání na dokončení diskové operace, nemůže přitom vyřizovat žádné další požadavky. Bylo by vhodné nějakým způsobem využít čas, kdy disk čte nebo zapisuje data. K tomu je potřeba tzv. asynchronní IO. Při použití asynchronního IO proces provede syscall pro čtení nebo zápis dat, tento syscall se okamžitě vrátí zpět do procesu a samotné čtení nebo zápis je prováděno paralelně při běhu procesu v userspace. Proces se pak nějakým způsobem dozví, že operace čtení nebo zápisu skončila.

Existuje standard POSIX async IO. Proces pošle asynchronní požadavek pomocí syscallů `int aio_read(struct aiocb *iocb)` nebo `int aio_write(struct aiocb *iocb)`. `struct aiocb` obsahuje obvyklé parametry, jako je handle, adresa v paměti a délka. `aiocb` obsahuje také místo, kam jádro vrátí návratový kód operace. Funkce `aio_read` a `aio_write` pošlou požadavek na čtení nebo zápis jádru a okamžitě se vrátí. Pomocí `int aio_return(struct aiocb *iocb)` je možno zjistit návratový kód (pokud operace ještě nedoběhla, funkce vrátí chybu `EINPROGRESS`). Pomocí `int aio_suspend(struct aiocb *iocbs[], int n_iocbs, struct timespec *timeout)` je možno čekat, než některý z asynchronních požadavků určených v poli neskončí nebo než nevyprší `timeout`. Probíhající asynchronní operaci je možno zastavit pomocí syscallu `aio_cancel`.

POSIX async IO má značné nedostatky[2]— není možno současně čekat na možnost čtení nebo zápisu do handlu (jako při `select` nebo `poll`) a na dokončení asynchronní operace. Dalším nedostatkem je syscall `aio_suspend`, který prochází všechny asynchronní požadavky. Složitost je úměrná počtu požadavků, takže tu vyvstává stejný problém jako u `select` a `poll`.

FreeBSD 4 i 5 má POSIX async IO (je nutno povolit `VFS_AIO` v konfiguračním souboru při kompilaci jádra; komentář u této volby varuje, že to není moc bezpečné). Většina IO subsystémů v jádře je psaná synchronně a očekává, že bude běžet v kontextu procesu.

[2] Je vidět, že je navrhovala standardizační komise a ne programátoři.

Pro korektní implementaci async IO by bylo třeba všechny tyto subsystémy kompletně přepsat. Aby je vývojáři jádra nemuseli přepisovat, udělali následující řešení — jádro spustí několik kernel threadů, z nichž každý může provádět jeden asynchronní požadavek. Po skončení vykonávání požadavku thread čeká na další.

FreeBSD přichází s rozšířením rozhraní — je zde nová funkce `int aio_waitcomplete (struct aiocb **iocbp, struct timeval *timeout)`, která počká na první dokončený požadavek a pointer na něj uloží na danou adresu. Tím se zabraňuje nutnosti specifikovat a procházet všechny požadavky u `aio_suspend`. Na FreeBSD je také možno na dokončení asynchronního IO čekat pomocí kqueue.

Linux nemá asynchronní IO. Knihovna GNU libc umí emulovat POSIX async IO pomocí syscallu `clone` — vyrobí nový thread a ten pak nechá provést požadavek. Výroba threadu je náročná operace, proto to není moc rychlé. V jádře Linuxu 2.5 jsou nějaké náznaky, že se vývojáři o implementaci async IO snaží, ale zatím mají pouze `aio` syscally, které dělají synchronní IO.

12.6. Řešení problému čekání na události na jiných systémech

Zde se krátce zmíním o tom, jak byl problém čekání na události řešen na jiných operačních systémech.

Většina komerčních Unixů (Solaris, IRIX, AIX, Digital Unix) mají asynchronní IO podle normy POSIX. Zpravidla je implementováno pomocí několika kernel threadů, které dělají synchronní IO.

Na Solarisu může proces otevřít `/dev/poll` a tím získá handle, který je funkčně podobný kqueue. Je možno na něm registrovat události nebo vybírat události, které nastaly. Množství událostí není tak velké jako u kqueue — u `/dev/poll` je možno registrovat jen možnost čtení nebo zápisu na nějaký handle.

Systém VMS používá AST neboli Asynchronous system trap. Kód procesu může kdykoli poslat na svůj vlastní ring nebo na ring s nižší úrovní privilegovanosti asynchronous system trap. AST je pointer na funkci a parametr, který této funkci bude předán. Až se systém dostane na úroveň privilegovanosti, na kterou bylo AST posláno, a pokud není vykonávání AST zamaskováno, je funkce s daným parametrem zavolána[3]Většina syscallů VMS má dvě verze — blokující verzi, poznáme ji tím, že název syscallu končí písmenem „W“, a neblokující verzi, se stejným názvem, ale bez „W“. Blokující verze syscallu počká, než se syscall dokončí, a pak se vrátí. Neblokující verze jako parametr dostane AST, vrátí se ihned, a zavolá danou AST, až byl syscall dokončen. Ukážeme si to na syscallu `sys$qio`, který se používá ke vstupu nebo výstupu na handle. Prototyp syscallu je `int sys$qio(unsigned event_flag, unsigned short handle, unsigned function,`

[3] Popis vypadá poněkud složitě — zjednodušeně to asi znamená tohle: jádro může poslat AST samo sobě a tato AST se ihned vykoná. Jádro může poslat AST uživatelskému procesu a tato AST se vykoná, až proces opustí jádro a vrátí se do userspace (je to obdoba unixového signálu). Proces může poslat AST sám sobě a tato AST se ihned vykoná. Ve skutečnosti je to trochu složitější, neboť VMS má celkem čtyři ringy — dva pro jádro a dva pro procesy.

`struct _iosb *iosb, void (*astaddr)(__int64 param), __int64 astparam, další parametry specifické pro dané zařízení a funkci...`). První parametr je event flag, který se má nastavit (popisem event flagů se zde nebudu zabývat — viz manuál k VMS), druhý parametr je handle, třetí parametr je kód funkce, co se má provést, čtvrtý parametr je pointer na místo v paměti, kam se uloží návratová hodnota a počet přenesených bytů, pátý parametr je funkce, která bude zavolána, až přenos skončí a šestý parametr je hodnota, která bude předána této funkci. Stejně tak v systému existuje funkce `sys$qiow`, která má shodné parametry s `sys$qio`, ale počká a vrátí se, až bude operace dokončena. Pomocí těchto asynchronních funkcí je možno jednoduše naprogramovat jednothreadový server, který bude paralelně zpracovávat větší množství požadavků. AST řeší jak problém současného čekání na větší množství událostí, tak asynchronní IO.

13. Závěr

Cílem této práce bylo zhodnotit klady a zápory operačních systémů Linux a FreeBSD v jednotlivých oblastech a objasnit algoritmy, které se v systémech používají.

FreeBSD 4 má menší latenci přerušení než Linux. Linux má jednoznačně lepší podporu víceprocesorových strojů než FreeBSD (ale víceprocesorové systémy s sebou přinášejí větší riziko chyb). Linux má pevně mapovanou paměť jádra, FreeBSD mapování paměti jádra vytváří dynamicky. Na 32bitových architekturách umí FreeBSD lépe pracovat s celými 4G adresního prostoru, nicméně 4G je maximální množství paměti pro FreeBSD. Linux na 32bitových architekturách limit pro maximální množství paměti nemá, ale občas musí zbytečně kopírovat data pomocí bounce-bufferů. Scheduler na Linuxu 2.4 a nižších měl velikou časovou složitost, na Linuxu 2.5 je nový scheduler s malou složitostí a s optimalizací pro SMP. Scheduler FreeBSD je kvalitativně někde mezi Linuxem 2.4 a 2.5. Scheduler na FreeBSD 4 je zcela nevhodný pro víceprocesorové stroje. Na FreeBSD 5 se chystá zcela nový scheduler se thready, které budou zčásti userspacové a zčásti řízené jádrem. Linux má obecně vzato jednodušší a čistší rozhraní k filesystému, FreeBSD je na tom trochu komplikovaněji — na druhou stranu FreeBSD umí pracovat s buffery libovolné velikosti; na Linuxu je velikost bufferu omezena velikostí stránky. Linux umí pracovat s větším množstvím filesystémů; některé jsou žurnálované. FreeBSD žurnálovaný filesystém nemá, ale integritu dat v případě výpadku zajišťuje technikou soft-updates. FreeBSD má jediný použitelný diskový filesystém UFS. V oblasti virtuální paměti dlouho vedlo FreeBSD, nicméně Linux se mu v několika posledních verzích 2.4 vyrovnává. Obecně vzato má Linux jednodušší správu virtuální paměti než FreeBSD — v některých typech zátěže je to výhoda, v jiných nevýhoda. Zero-copy TCP byla dlouho veliká výhoda FreeBSD nad Linuxem; nicméně v posledních verzích bylo zero-copy TCP přidáno i do Linuxu 2.4. Pro efektivní čekání na více událostí Linux implementuje POSIX realtimové signály — nicméně jejich použití je značně komplikované, neboť se signály mohou kdykoli ztrácet. FreeBSD realtimové signály neumí, používá nestandardní rozhraní `kqueue`. Toto rozhraní je velmi flexibilní a snadno použitelné. Linux 2.5 má rozhraní `epoll`, které odstraňuje nevýhody realtimových signálů, nicméně není tak universální jako `kqueue`. FreeBSD umí dělat async IO pomocí kernel threadů; Linux async IO neumí a linuxová libc ho emuluje pomocí uživatelských threadů.

Neexistuje žádné universální rozhodnutí, zda je lepší Linux, nebo FreeBSD — každý systém má v různých oblastech svoje klady a zápory a ty byly v této práci popsány. Jak bylo vidět, pro některé typy úloh je vhodnější použít Linux, pro jiné FreeBSD.

14. Literatura

[1] Zdrojové kódy Linuxu
ftp://ftp.cz.kernel.org/

[2] Zdrojové kódy FreeBSD
ftp://ftp.freebsd.cz/

[3] Linux-kernel mailing list
http://www.uwsg.iw.edu/hypermail/linux/kernel/

[4] FreeBSD mailing listy
http://www.freebsd.org/mail/

[5] Dennis M. Ritchie — The Evolution of the Unix Time-sharing System
http://cm.bell-labs.com/cm/cs/who/dmr/hist.html

[6] Jeff Bonwick — The Slab Allocator
http://luthien.nuclecu.unam.mx/~miguel/bonwick.ps

[7] Linux — Dokumentační projekt
Computer Press, 1998
http://www.cpress.cz/knihy/linux/

[8] Matthew Dillon — Design Elements of the FreeBSD VM System
http://www.daemonnews.org/200001/freebsd_vm.html

[9] Rik van Riel — talk at OSDN summit
http://ftp.nl.linux.org/misc/mp3_osdn_8summit.mp3

[10] Rik van Riel — Page replacement in Linux 2.4 memory management
http://www.surriel.com/lectures/linux24-vm.html

[11] Andrea Arcangeli — Le novita' nel Kernel Linux
http://meeting.pluto.linux.it/atti/pluto-dec-pub-0_tar.gz

[12] Antonis Argyriou — The FreeBSD scheduler
http://www.geocities.com/kejriwal_nidhi/ece6277classpr/Downloads/
FreeBSDscheduler.pdf

[13] Martin Mareš — Wonderful World of Linux 2.5
ftp://atrey.karlin.mff.cuni.cz/pub/local/mj/papers/
linux25-slt2002.ps.gz