Západočeská univerzita
FAKULTA APLIKOVANÝCH VĚD

Okruhy otázek ke státní závěrečné zkoušce z předmětu
Systémové programování (SP)

Operační systémy (OS)
Paralelní programování (PPR)
Formální jazyky a překladače (FJP)
Výkonnost a spolehlivost čísl. systémů (VSP)

Studijní program:	3902	Inženýrská informatika
Obor:	2612T025	Informatika a výpočetní technika – Softwarové inženýrství
	3902T031	Softwarové inženýrství

Akademický rok: 2005/2006

Obsah:

1 Operační systémy: Funkce operačního systému, rozhraní operačního systému, struktura operačního systému, mikrojádro

Operační systém je sada programů (software) umožňujících co nejefektivnější využití počítače. Základním úkolem operačního systému zabezpečit podporu a zpracování aplikačních programů.

1.1 Funkce operačního systému

- správce zdrojů - resource manager
- virtuální počítač - virtual machine

Správce zdrojů - Zdroje jsou I/O zařízení, soubory, procesor, paměť apod. Operační systém vlastní jednotlivé systémové zdroje - přiděluje a odebírá je jednotlivým procesům.

Virtuální počítač - Operační systém skrývá detaily ovládání jednotlivých zařízení (transparentnost), definuje standardní rozhraní pro volání systémových služeb. Programátor se může věnovat vlastní úloze a nemusí znovu programovat I/O operace. Program může díky "odizolování" od konkretních zařízení pracovat i se zařízeními, o kterých jeho autor v době vytváření programu neměl ani ponětí (program se o ovládání I/O nestará).

1.2 Rozhraní operačního systému

1.2.1 Uživatelské rozhraní OS

- je tvořeno souborem aplikačních (uživatelských) programů
  - interpret příkazů (shell)
  - příkazy pro práci se soubory a adresáři
  - filtry (sort)
  - nástroje pro vývoj programů (editory, překladače,…)
  - správa systému
  - různé
- ovládání
  - příkazový řádek
  - okno, GUI

1.2.2 Programové rozhraní OS

- je tvořeno souborem systémových volání, které poskytuje jádro OS
- instrukční soubor je tedy rozšířen o vykonání dalších operací – rozšířený stroj
  - na vykonání systémových volání využívá všechny instrukce CPU
  - některé z nich nemůžou používat aplikační (uživatelské) programy – privilegované instrukce
    - práce s některými registry – PC, stavový registr
    - práce s HW
- CPU musí „vědět“, jestli vykonává instrukci aplikačního programu nebo instrukci jádra

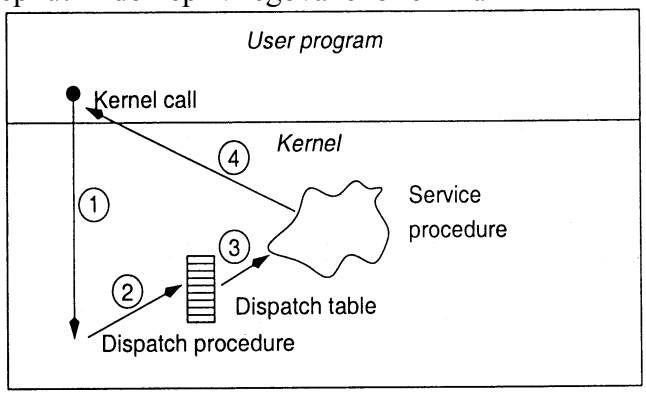
Řešení:

- CPU musí mít možnost vykonávat instrukce v různých stavech, režimech – např. procesory Intel 80x86 mají čtyři různé stavy
- privilegovaný režim (režim jádra (*kernel mode*), režim supervizoru)
  - přepnutí do privilegovaného režimu
    - program žádá systémové volání
    - výjimka (exception)
    - přerušení od zařízení
- neprivilegovaný režim (uživatelský režim)
  - přepnutí do neprivilegovaného režimu, při návratu z privilegovaného, instrukcí, která je privilegovaná
  - všechny aplikační programy, tedy i programy tvořící uživatelské rozhraní, se vykonávají v neprivilegovaném režimu

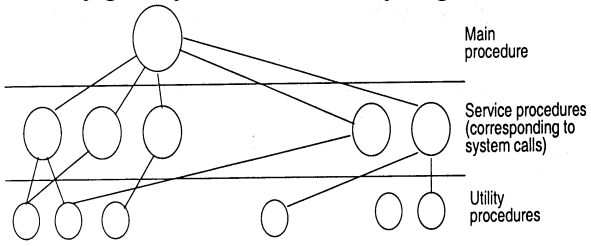
1.3 Struktura operačního systému

- Monolitické systémy
  - pro jednotlivé funkce jsou definovány moduly
  - modul může volat jakýkoli jiný modul
  - všechny moduly jsou spojeny do vykonatelného souboru s operačním systémem

- Systémové volání (služba jádra)
  - volání vstupního bodu jádra OS s přepnutím do privilegovaného režimu
  - zjištění čísla požadované služby
  - volání obslužné procedury
  - návrat s přepnutím do neprivilegovaného režimu



- Model struktury monolitického systému
  - hlavní program, který spouští obslužnou proceduru
  - množina obslužných procedur pro systémová volání
  - podpůrné procedury pro vykonání obslužných procedur



- Monolitické systémy
  - mají tendenci extrémně narůstat
  - monolit akumuluje moduly, které by potenciálně mohli být potřebné

- těžce se ladí

- Příklady: Linux, MacOS, Windows

Pokračování systémy založenými na mikrojádře v další kapitole.

1.4 Mikrojádro

- Vrstvené systémy
  - hierarchie vrstev poskytujících služby
  - programy vyšší vrstvy využívají služeb nižších vrstev
  - holý počítač je nejnižší vrstva
  - aplikační program je nejvyšší vrstva
  - princip vrstev umožňuje systematickou tvorbu programů a jejich testování
  - princip vrstev je možné použít pro monolitický model i pro systémy založené na mikrojádře

- Mikrojádro
  - vrstva nad holým strojem, která obsahuje minimální množinu abstrakcí, tak aby ostatní funkce OS mohli být implementovány nad ním
  - tyto funkce OS nemusí být vykonávány v privilegovaném režimu
  - jenom mikrojádro musí být vykonáváno v privilegovaném režimu
  - typická množina abstrakcí implementována mikrojádrem (zdola nahoru)
  - přerušení, nízkoúrovňový V/V
  - vlákna
  - správa paměti (JavaOS)
  - meziprocesovou komunikaci
  - procesy
  - ostatní funkce - soubory, adresáře, síťové služby - jsou programy vykonávané v uživatelském režimu

2 Případová studie OS Linux

2.1 Stručná historie UNIXu, LINUXu a GNU

- 1965-1969 : Bell Telephone Labs (BTL), AT&T, MIT, GE - Multics
- 1969 Ken Thompson a Dennis Ritchie na PDP-7 vytvořili základ něčeho, co se později použilo jako základ systému na zpracování dokumentů v AT&T.
- 1971 - 1. verze (V1), na PDP-11/70, v assembleru. Brian Kernighan navrhl název Unics -> UNIX
- 1973 - V4. Přepis do jazyka C - jedna z nejvýznamnějších událostí v historii OS: portabilita systému na jiné architektury.
- 1974 Ritchie s Thompsonem - zpráva o systému UNIX v CACM (Communications of the Association for Computing Machinery)
- 1975 - V6 je první mimo BTL - systém bezplatně předán univerzitám. V tomto období vzniká také první univerzitní větev systému, 1.xBSD (Berkeley Software Distribution)
- 1979 - V7: dnes známá základní sada příkazů - shell, překladač C, uucp,...
- 1982 - 1. komerční systémem AT&T : UNIX System III
- IBM, DEC, Hewlett-Packard a další - devadesátá léta - konsorcium OSF (Open Software Foundation)
- 1983 Richard M. Stallman (autor EMACS a Lisp) začíná projekt GNU (= Gnu's Not Unix) na MIT. První GPL (General Public Licence) - přesně specifikuje podmínky, za jakých je možno šířit, používat a vyvíjet tzv. "svobodný" software - software, jenž uživateli zaručuje tato práva:
  - spustit program za jakýmkoliv účelem.
  - modifikování programu podle jeho potřeb. (Aby tato svoboda byla realizovatelná v praxi, je nutno mít přístup ke zdrojovému kódu, neboť vytváření změn v programu je nesmírně obtížné, není-li k dispozici zdrojový kód.)
  - redistribuování kopií a to jak zadarmo, tak za poplatek.
  - distribuci modifikovaných verzí programu, aby komunita mohla mít prospěch z jeho vylepšení.

  Hlavní myšlenkou GPL je dát každému povolení ke spouštění, kopírování, modifikaci programu a šíření modifikovaných verzí - ne však povolení přidávat k nim vlastní omezení. Takto jsou rozhodující svobody, které definují "svobodný software" zaručeny pro každého, kdo má kopii; stávají se nezcizitelnými právy. Mj. to znamená, že co je vytvořeno GPL (nebo GNU) programem (tj. jakýkoli produkt vytvořený za využití software licencovaného podle GNU GPL licence), to na sebe automaticky převádí GPL
- 1984 AT&T zveřejňuje System V Interface Definition (SVID) - standartizace Unixových rozhraní. Pět evropských výrobců počítačů zakládá X/Open: Bull, ICL, Siemens, Olivetti, a Nixdorf.
- 1985 ------//----- Free Software Foundation (nadace pro vývoj volného SW) - EMACS, překladač C/C++/Objective C/Fortran/Pascal (50 platforem)
- 1988 - IBM, DEC, HP, a další zakládají Open Software Foundation (OSF) jako soupeře sdružení AT&T/Sun, chtějí použít jádro AIX. Jako odpověď na OSF je

založeno UNIX International (UI) jako mezinárodní consortium uživatelů Unix System V UNIX. Úzká spolupráce s AT&T s cílem podpory otevřených systémů a ovlivnění dalšího vývoje.

- 1989 - Programovací jazyk C je standardizován ANSI pod označením X3.159.1989 jako mezinárodní standard ISO/IEC 9899:1990.
- 1990 - UNIX International vydává System V Release 4 (SVR4) který je sjednocením Systemu V, BSD a XENIXu (SCO).
- 25.8.1991 - neznámý finský student Linus Benedict Torvalds zaslal do diskuzní skupiny comp.os.minix příspěvek s předmětem "What would you like to see most in minix?". O tom, že vyrábí free operační systém - nic komerčního a velkého, pouze jen tak ze zábavy.
- standard POSIX (Portable Operating System Interface)
- 1994 - vydána verze 4.4BSD. Patrick Volkerding sestavuje distribuci Slackware. Linux je portován na ne-Intel platformy (MIPS, Alpha,...). Vydány verze FreeBSD and NetBSD, pod BSD licencí. Novell v červnu kupuje USL od AT&T, v říjnu přenechává ochrannou známku UNIX sdružení X/Open (mezinárodní organizace pro standardizaci takzvaných "otevřených systémů")
- 1996 - X/OPEN a OSF se spojují v Open Group. Je dokončen Linux 2.0.
- 1997 - Single Unix Specification, V2, vydána. Linux se začíná stávat operačním systémem ISP (Internet Service Provider).
- 1998 - uveřejněno označení UNIX98, zahrnuje 3 kategorie UNIXů: základ, pracovní stanici (workstation), server. Open Source hnutí nabírá otáčky. Tisk objevuje Linux a Open Source hnutí. Oracle, Informix, IBM, Compaq a další ohlašují svou podporu Linuxu.
- 1999 - Průmysl se začíná zajímat o Linux a UNIXové produkty. Hlavní komerční SW vývojáři začínají vydávat verze pro Linux (SAP oznamuje SAP/R3 pro Linux. Apple vydává Mac OS X založený na jádře Mach,..)
- 2000 - Hlavní komerční prodejci HW (Compaq, IBM, Dell, SGI, Fujitsu) začínají prodávat desktop a laptop počítače s předinstalovaným Linuxem. Linux je portován na IBM S/390. Lockheed Martin Corp. užívá Linux NetworX clusterovou technologii na analýzu U.S. Navy letadel. IBM investuje více než 200 miliónů $ do řady Linuxových iniciativ v Evropě během následujících čtyř let. HP uveřejňuje HP-UX 11i, plně 64 bitový Unix kompatibilní s Linuxem.
- 2001 - uveřejněn Linux v2.4. Sony oznamuje portaci Linuxu na PlayStation 2. Nokia adoptuje Linux pro vývoj aplikací pro svůj Media Terminal home entertainment system. HP adopts Debian as the selected development platform for Linux work at HP. IBM oznamuje nový supercluster na Linuxu, instalovaný v National Center for Supercomputing Applications (NCSA), s výkonem 1 trillion výpočtů za vteřinu, Cluster z 160 nových IBM na Itanium CPU založených systémech bude nejvýkonnější Linuxový supercluster na akademii.
- 25.8.2001 - 10.-té výročí Linuxu. Za deset let vývoje je linux nyní v úplně jiných sférách, než byl na začátku - na linuxu si můžete pustit grafické rozhraní, přehrávat mp3 nebo video, funguje vám televizní karta, máte na výběr z několika browserů, vývojových prostředí, nepřeberné množství síťového softwaru, podporuje sambu, joliet a jiné microsoftí věci. Linux se stal úspěšným konkurentem jak na serverech, kde úspěšně nahrazuje starší a hlavně drahé unixy, tak na desktopových stanicích, kde začíná úspěšně vytlačovat MS Windows (je otázkou, jak rychle a zda vůbec, ale za

předpokladu, že linux na svém domácím stroji má stále více a více lidí, je pravdivost této věty snad jen otázkou času ;-)), a jedním z hlavních faktorů, proč tomu tak je, je jeho nízká cena a velmi rychlý vývoj.

- 9.9.2001 1:46:40 UTC - 1e9 (=miliarda) vteřin od 1.1.1970 0:0:0 - data považovaného za počátek éry Unixu a braného jako čítač hodin a "časové razítko" (timestamp) jevů v Un*x systémech. Viz příkaz:
- date '+%s' -d "9-Sep-2001 1:46:40 UTC"
- 2002 - Free Standards Group uveřejňuje LSB 1.1 (včetně plné množiny společných API a vývojového balíku), a Li18nux (internacionalizační příručku), dva nástroje pro zajištění toho, že všechny Linux aplikace mohou běžet na libovolné verzi která je kompatibilní s Linux Standard Base (LSB). IBM přijímá Red Hat Linux Advanced Server pro všechny své servery a mainframe.

2.2 Charakteristika operačního systému GNU/Linux

- **OS Unixového typu** - filozofie, procesy, uživatelé, souborový systém, základní programy a další věci jsou shodné s Unixovými standardy.
- **Čistě 32/64 bitový OS** - Linux od počátku byl psán jako 32-bitový OS a dnes podporuje řadu 64-bitových architektur - první byly procesory DEC Alpha, nyní též Intel64, 64-bit procesory MIPS a SPARC a 64-bit z/Architektura na S390 fy Intel. Pozn.: GNU C knihovna (GLIBC) je již částečné portována i na 128-bitové architektury.
- **Víceúlohový OS** - jeden člověk může mít spuštěno několik programů současně
- **Víceuživatelský OS** - více lidí může současně pracovat na jednom fyzickém počítači. OS uživateli vytváří virtuální prostředí tvářící se, jako by měl počítač sám pro sebe: nikdo nebude bez jeho povolení číst jeho soubory, nikdo nebude zasahovat do běhu jeho programů, bude moci používat periferní jednotky počítače (tiskárny, vstupní jednotky,..) atd.
- **Víceprocesorový OS** - SMP podle dané architektury - podporováno až 64 procesorů. OS zaručuje rovnoměrné využití procesorů jednotlivými procesy
- **Preemptivní OS** - žádná úloha si nemůže "přivlastnit" a zablokovat systém; systém po určité době přidělení sám odebírá úlohám procesor(y). Úloha si vůbec nemusí uvědomovat existenci střídání se o procesor.
- **Real-time OS** - kromě normálních typů procesů (-úloh) jsou podporovány i real-time procesy. Jsou jinak plánovány a mají vyšší prioritu než všechny ostatní procesy.
- **Všestrannost nasazení** - používá se od tzv. zapouzdřených (angl. embedded) systémů (specializovaná nasazení většinou na mini HW a speciálních periferiích - řídící systémy, roboti, telefony ap. Díky své modularitě Linux pracuje v podmínkách, kde jiné OS bídně hynou) přes PDA (Personal Digital Assistent), servery (souborové, datové-SQL, síťové, tiskové aj.) po grafické pracovní stanice s X-Window systémy. Stejně dobře pracuje v jednouživatelském, víceuživatelském textovém i víceuživatelském grafickém režimu.
- **Svobodný SW** - základem Linuxu je volně šiřitelný SW /vč. zdrojových kódů/. Kdokoliv může zdrojové kódy volně používat, upravovat a šířit - viz GNU GPL licence. Jádro i aplikační programy jsou vyvíjeny a spravovány tisíci nadšenci po celém světě komunikujícími po Internetu. Současně je ale na Linux portováno a pro něj vyvinuto mnoho komerčních programů, zpravidla za daleko nižší cenu než odpovídající verze pro komerční Unixové systémy.
- **Nejrychleji se rozvíjející OS** - za 10 let existence GNU/Linux vyrostl od původní verze (na i386, se souborovým systémem Minix a z programů pouze překladač C a

shell) k dnešnímu stavu (jádra 2.6.x, zdrojové soubory zabírají již přes 200MB) - podpora více než 20 HW architektur, SMP, několika desítek souborových systémů a řada dalších vlastností v hlavním vývojovém stromu jádra. K tomu je nepřeberná řada jádro obalujících GNU systémových, uživatelských a dalších programů, několik X-Window manažerů,...

2.3 Filozofie operačního systému Unix

Unix byl OS určený primárně na zpracování textu, a většina komponent pracuje s textovým vstupem a produkuje textový výstup. Od počátku byly programy psány s několika základními principy:

provádět právě jednu věc a tu dělat dobře
zaručit jejich vzájemnou spolupráci
povely a data programy přijímají v textové podobě
jejich výstupy jsou též v textové podobě a ve formě vhodné pro další zpracování jako vstup jiných programů.

Od opuštění assembleru jsou programy psány již výhradně v jazyce C. I když je programování v C bez ohledu na platformu v mnoha směrech stejné, je třeba říci, že unixoví vývojáři nahlíží na vývoj programů a systémových nástrojů specifickým způsobem. Operační systém Unix/Linux podporuje určitý styl programování a Unixové programy a systémy sdílí následující charakteristiky:

Jednoduchost
Valná většina utilit pro operační systém Unix je velmi jednoduchých a v důsledku toho také malých a snadno pochopitelných. Je dobré si osvojit techniku zvanou KISS (Keep It Small and Simple - Snaž se to udržet malé a jednoduché). U větších a složitějších systémů je vyšší pravděpodobnost většího množství komplexních chyb jejichž ladění může být obtížné.

Zaměření
Vždy je lepší vytvořit program, který provádí dobře jen jeden úkol. Program přecpaný nejrůznějšími funkcemi se obtížně používá a udržuje. Jednoúčelové programy se snáze vylepšují, když se objeví lepší algoritmy nebo rozhraní. V systému Unix jsou v případě potřeby malé utility často kombinovány tak, aby prováděly náročnější úkoly, místo aby se programátoři snažili předvídat potřeby uživatelů za pomoci jednoho velkého programu.

Znovu použitelné komponenty
Je užitečné dát jádro aplikace k dispozici v podobě knihovny. Dobře dokumentované knihovny disponují jednoduchým ale flexibilním rozhraním, mohou ostatním lidem pomoci při vývoji různých variací nebo při aplikaci postupů v nových oblastech. Příkladem budiž databázová knihovna dbm, což je spíše než jeden program pro správu databáze sada znovu použitelných funkcí.

Filtry
Spoustu unixových aplikací lze využít jako filtry. To znamená, že mohou převádět vstup na výstup jiného typu. Systém Unix/Linux poskytuje nástroje, které umožňují vzájemnou kombinací jiných unixových programů novými a neotřelými způsoby vytvářet poměrně složité aplikace. Samozřejmě, že je tato možnost opětovného použití dána právě zmíněnými vývojovými metodami.

Otevřené formáty souborů
Nejúspěšnější a nejoblíbenější unixové programy používají konfigurační a datové soubory, které mají podobu textových ASCII souborů. Uživatelé tak totiž mohou měnit a prohledávat

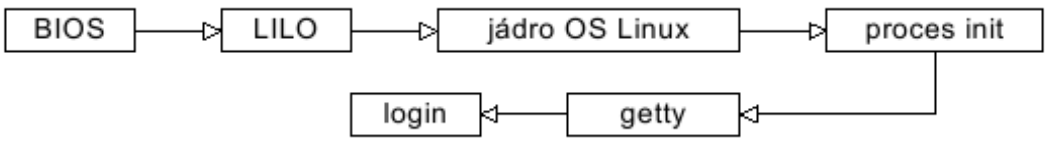
konfigurační nastavení pomocí standardních nástrojů a zároveň vyvíjet nástroje, které budou při práci s datovými soubory používat nové funkce.

Flexibilita

Nelze dopředu předvídat, jak důmyslní uživatelé budou program používat. Je proto dobré při programování zachovávat co největší flexibilitu. Dobří programátoři se vyhýbají svévolným omezením velikostí polí nebo počtu záznamů. Je-li to možné, jsou programy psány tak, aby mohli pracovat v síti stejně dobře, jako na lokálním počítači. Nikdy si nemyslete, že jste vzali do úvahy vše, do může uživatel chtít udělat.

3 Zavedení operačního systému

1. BIOS
   - provádí příslušné testy a nastavení
   - vybere zaváděcí jednotku (FDD, CD-ROM, HDD – vybere konkrétní oblast)
   - načte první sektor (zaváděcí sektor) (u HDD MBR), ve kterém je uložen zavaděč (program pro spuštění operačního systému)
2. Zavaděč – LILO
   - LILO umožňuje zvolit zavedení Linuxu nebo jiného operačního systému
   - načte jádro operačního systému do paměti a spustí ho
3. Jádro OS Linux
   - jádro Linuxu se instaluje v komprimovaném tvaru (před samotným zavedením se samo dekomprimuje)
   - detekuje hardware a odpovídajícím způsobem nastaví ovladače zařízení
   - průběžně vypisuje zprávy o nalezených zařízeních
   - připojí kořenový svazek pro čtení a provede kontrolu souborového systému
   - spustí na pozadí proces init z adresáře /sbin, program init se spouští vždy jako proces 1 (pid=1)
4. Proces init
   - proces init se konfiguruje pomocí souboru /etc/inittab
   - inicializuje operační systém
   - je spuštěn po celou dobu běhu operačního systému a ošetřuje některé události
   - přivlastňuje si osiřelé procesy (pokud rodičovský proces ukončí svou činnost dříve než jeho potomek, vzniká sirotek, který se automaticky stává potomkem procesu init)
   - úklid v adresáři /tmp
   - start různých služeb tz. Démonů
   - nakonec spustí program getty pro terminály a virtuální konzolu



4 Proces, systémová volání

4.1 Proces

- **program** = vykonatelný soubor
- **proces** = jedna instance vykonávaného programu

Unix souběžně (simultaneously) se může vykonávat mnoho procesů (šachový velmistr) může se vykonávat mnoho instancí jednoho programu (např. pogramu kp pro kopírování souborů)

4.2 Proces v UNIXu

- proces je jednotka (entita), která vykonává programy a poskytuje prostředí pro jejich vykonávání
- adresový prostor + počítadlo instrukcí
- proces je základní jednotkou plánování (*scheduling*)
- procesor vykonává v jednom okamžiku nejvíc jeden proces
- soutěží a vlastní prostředky
- požadují vykonání služeb jádra
- Systémová volání pro procesy
  - vytvoření procesu
    `pid = fork();`
  - vytvoří se (téměř) identická kopie volajícího procesu
  - adresový prostor je kopie adresového prostoru volajícího programu a vykonává se stejný program
  - vytvořený proces má svou kopii deskriptorů souborů, které odkazují na stejné soubory
  - volající proces rodič
  - vytvořený proces potomek
  - každý proces (kromě prvního má svého rodiče)
  - rodič může mít více potomků
  - návrat ze systémového volání (fork) na stejné místo
  - jak je rozeznáme ?
    - jádro identifikuje procesy číslem procesu, které se nazývá identifikátor procesu (*process identifier* PID)
    - návratová hodnota **pid** bude ve volajícím procesu PID vytvořeného potomka a v potomkovi bude nula
    - program může obsahovat kód rodiče i potomka

```
main() {
      /*kód rodiče*/
      pid=fork();
      if (!pid) {
            /*kód potomka*/
      }
      if (pid) {
            /*kód rodiče*/
      }
}
```

- častěji, v nově vytvořeném procesu se vykoná nový program voláním některého tvaru služby `exec`
  - `kp` – název souboru, který obsahuje vykonatelný program pro kopírovaní souborů

```
main(int argc, char *argv[]) {
     int stav;
     if (fork == 0)
          execl("kp", "kp", argv[1], argv[2], 0);
     wait(&stav);
     printf("kopirovani skonceno");
}
```

- původní program je v paměti přepsán a potomek nepokračuje vykonáváním starého programu, ale potomek se vrátí z volání s počítadlem instrukcí nastaveným na první vykonatelnou instrukci nového `programu`
- čekání na skončení potomka
  `pid = wait (stav_adresa);`
  - **stav_adresa** je adresa celočíselné proměnné, která bude obsahovat koncový stav procesu
- ukončení procesu
  `exit(stav);`
- C programy volají `exit` při návratu z funkce `main`

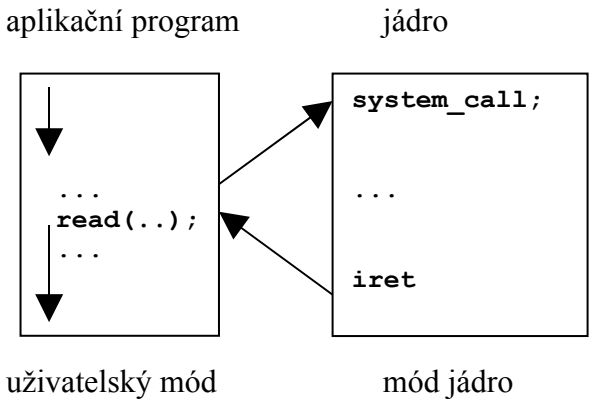
4.3 *systémová volání*

- `fork(),open(), write(), ...`
- je jich hafo:)
- práce s diskem, práce s programy, spouštění programů, ...
  - definováné POSIXem
  - MS Windows nesplňují (při použití POSIX subsystému nelze používat WIN32 API), emulace POSIXu - problémy s forkem.

5 Jádro a proces, stavy procesu, oprávnění uživatele, implementace procesu

5.1 Jádro a proces

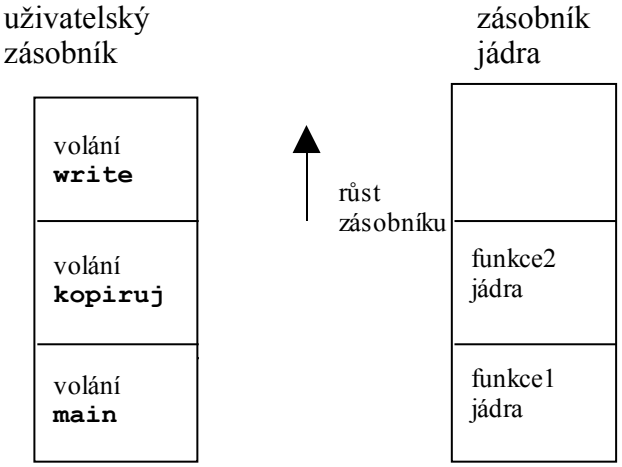
- co je jádro? je jádro proces?



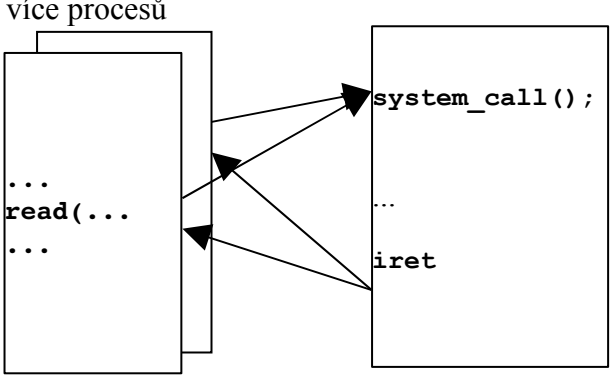
- vykonávaný program – proces = posloupnost instrukcí aplikačního (uživatelského) programu a jádra
- jádro – speciální program zavedený do hlavní paměti při startu systému přímo vykonávaný HW

- virtuální adresový prostor procesu
  - adresový prostor procesu (uživatelský)
  - systémový prostor (jádra)

- proces v uživatelském módu má přístup ke svému adresovému prostoru, k systémovému prostoru voláním sytem_call()
- jádro má přístup k adresovému prostoru procesů
- proces používá dva zásobníky
  - uživatelský
  - jádra



- více procesů

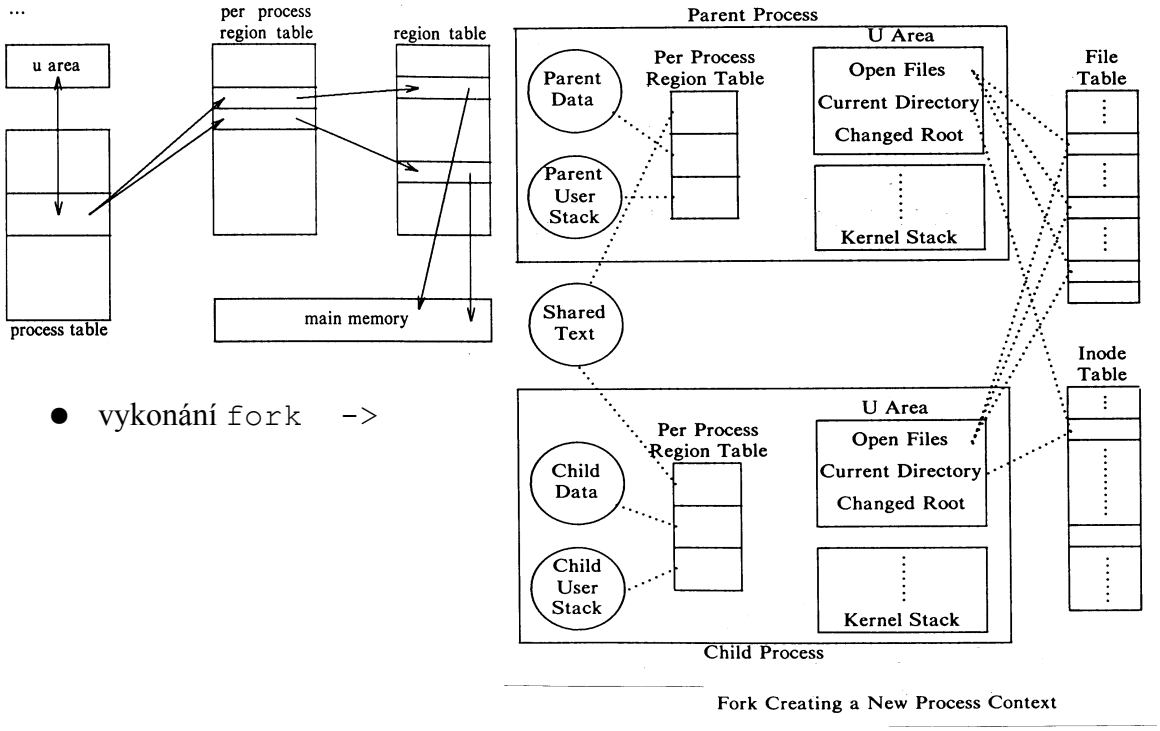


- jádro je reentrantní
- každý proces má svůj zásobník jádra, často v adresovém prostoru procesu – chráněný, spravovaný jádrem
- každý proces má položku v tabulce procesů proc záznam a u (*user*) oblast
- u oblast – údaje potřebné když je proces vykonávaný
  - tabulku deskriptorů souborů otevřených souborů
  - okamžitý adresář
  - kořenový adresář
  - často zásobník jádra procesu
  - ...

  v adresovém prostoru procesu – chráněná, spravovaná jádrem

5.1.1 proc záznam

- tabulka oblastí (*region*) procesu



Fork Creating a New Process Context

- vykonání `fork` ->

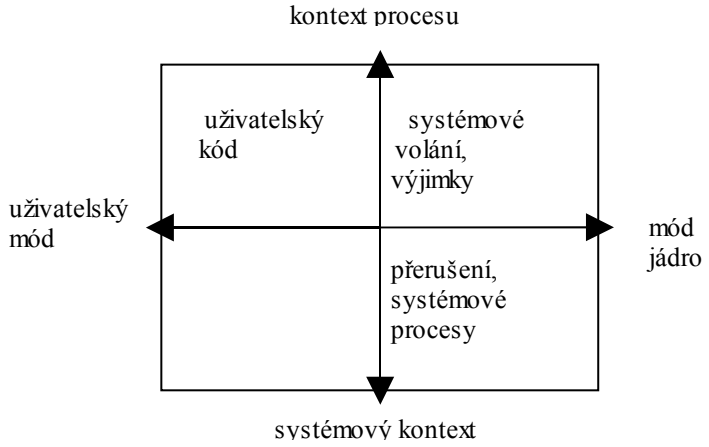
5.1.2 Kontext procesu - jeho stav

- uživatelský adresový prostor
  - text (vykonatelný kód)
  - data
  - uživatelský zásobník
- řídící informace
  - u oblast
  - proc záznam
  - zásobník jádra
- registry (HW kontext)
  - počítadlo instrukcí
  - ukazatel zásobníku
  - stavové slovo procesoru
    - mód
    - úroveň přerušovací priority
    - přetečení
    - ...
  - registry pro správu paměti
  - registry jednotky pohyblivé čárky

5.1.3 jádro

- vykonává služby
- zpracovává výjimky (pokus dělit nulou, přetečení uživatelského zásobníku, ...)
- zpracovává přerušení od periferních zařízení
- vykonává systémové procesy (správa paměti, přepočítávaní priorit procesů

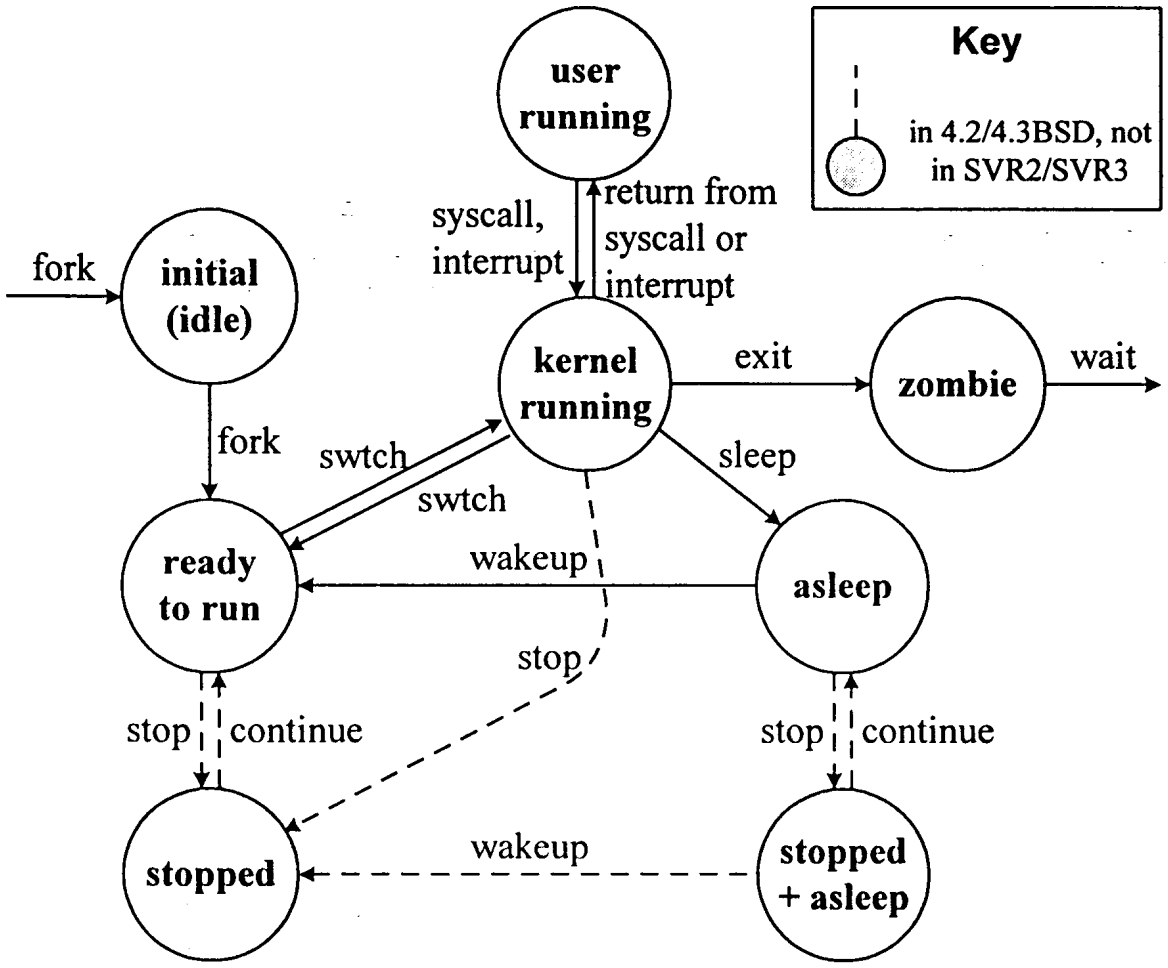
- jádro pracuje
  - v kontextu procesu
  - v procesu v systémovém kontextu



- **uživatelský kód** – přístup jenom k (uživatelskému) adresovému prostoru procesu
- **systémové volání, výjimky** – přístup k uživatelskému i systémovému adresovému prostoru
- **přerušení, systémové procesy** – přístup jenom k systémovému adresovému prostoru

5.2 Stavy procesu

v každém okamžiku se proces nachází v nějakém definovaném stavu, přechody mezi jednotlivými stavy znázorňuje diagram stavových přechodů



Process states and state transitions.

- **začáteční** (*initial/idle*) – fork začal vytváření procesu
- **připraven na vykonání** (*ready to run*) – proces čeká až bude naplánován na přidělení procesoru
- **běžící v jádře** (*kernel running*) – byl naplánován, vykoná se přepnutí kontextu, procedura jádra `swtch()` - uloží HW kontext do registrů
- **běžící uživatelsky** (*user running*) – může přejít do stavu běžící v jádře v důsledku volání služby jádra nebo přerušení, po skončení obsluhy se vrátí
- **spící** (*asleep*) – při vykonávání systémového volání se může stát, že je nutno čekat na nějakou událost nebo prostředek, proces (v jádře) zavolá proceduru `sleep()`, když událost nastane, jádro vzbudí proces a proces se stane připraven na vykonání a po naplánování pokračuje obsluha systémového volání ve stavu běžící v jádře

- **připraven na vykonání** se může stát je-li běžící po uplynutí přiděleného časového kvanta, vykoná se preempce běžícího procesu a to ve stavu běžící uživatelsky nebo při návratu do něj, jádro je nepreemptivní
- přerušení se může vyskytnout i ve stavu běžící v jádře, kdy po skončení obsluhy proces zůstane ve stavu běžící v jádře
- proces končí voláním `exit` anebo v důsledku signálu, přechází do stavu mátoha (*zombie*), dokud rodič nevykoná `wait`
- do stavu zastaven (*stopped*) nebo spící a zastaven přejde proces po stop signálech
- `SIGSTOP` zastav proces
- `SIGTSTP` CTRL-Z
- `SIGTTIN` tty čtení procesu v pozadí
- `SIGTTOU` tty psaní procesu v pozadí
- signál `SIGCONT` převede proces do stavu připraven na vykonání nebo do stavu spící

5.3 Oprávnění uživatele (user credentials)

- každý uživatel má v systému identifikován číslem
- identifikátor uživatele (*user identifier*) UID
- obdobně identifikátor skupiny (*group identifier*) GID
- identifikátory ovlivňují vlastnictví vytvářených souborů, používají se na kontrolu přístupových práv a kontrolu zasílaní signálů jiným procesům
- procesy dědí oprávnění
- privilegovaný uživatel *superuser* UID 0, GID 1

- každý proces má
  - reálný UID (*real UID)* - identifikuje reálného uživatele a ovlivňují právo posílat signály
  - efektivní UID *(effective UID)* - ovlivňují vlastnictví souborů a přístup k souborům
  - uložený nastavovací UID *(saved set UID)*
- proces změní efektivní UID
  - vykoná-li exec programu s nastaveným bitem SUID
  - systémovým voláním setuid

- bit SUID, nastavení UID (set UID), je jeden z bitů „přístupových“ práv k souboru
- je-li nastaven, efektivní UID a uložený nastavený UID se nastaví na ID vlastníka souboru

- `setuid(uid)`
  - je-li okamžitý efektivní UID procesu privilegovaný uživatel, potom reálný UID, efektivní UID a uložený nastavovací UID se nastaví na uid
- jinak `setuid(uid)` nastaví efektivní UID na hodnotu `uid`, je-li `uid` rovno reálnému UID nebo uloženému nastavovacímu UID, reálný UID a uložený nastavovací UID se nezmění

	setuid(e) euid==0	setuid(e) euid!=0
reálný UID	**e**	nezměněn
efektivní UID	**e**	**e**
uložený nastavovací UID	**e**	nezměněn

- následující program po přeložení vlastní „on“, UID == 800, má nastavený bit SUID, právo vykonávat mají všichni
- uživatel „on“ vlastní soubor „on“ s právem čtení jenom pro vlastníka
- uživatel „já“ , UID == 500 vlastní soubor „ja“ s právem čtení jenom pro vlastníka

```
main() {
     int uid, euid, fdja, fdon;

     uid = getuid();   /*reálný*/
     euid = geteuid(); /*efektivní*/
     printf("uid %d euid %d\n", uid, euid);

     fdja = open("ja",O_RDONLY);
     fdon = open("on",O_RDONLY);
     printf("fdja %d fdon %d\n", fdja, fdon);

     setuid(uid);
     printf("uid %d euid %d\n", uid,euid);

     fdja = open("ja",O_RDONLY);
     fdon = open("on",O_RDONLY);
     printf("fdja %d fdon %d\n", fdja, fdon);

     setuid(uid);
     printf("uid %d euid %d\n", uid,euid);
}
```

- vykonání uživatelem „ja“:

```
uid 500 euid 800
fdja -1 fdon 3
uid 500 euid 500
fdja 4 fdon -1
uid 500 euid 800
```

- vykonání uživatelem „on“:

```
uid 800 euid 800
fdja -1 fdon 3
uid 800 euid 800
fdja -1 fdon 4
uid 800 euid 800
```

- Je-li euid == 0 změní se reálný UID, efektivní UID i uložený nastavovací UID.

- **login** po úspěšném přihlášení uživatele nastaví reálný UID, efektivní UID i uložený nastavovací UID na UID uživatele a vykoná **exec shell**
- Chce-li uživatel změnit své heslo, nemůže tak učinit přímo.
- Vykoná program **passwd,** který má nastavený bit SUID, a který vlastní superuser.

Obdobně s GID

- Rodina volání:
  - *getegid*() , *geteuid*() , *getgid*() , *getuid*() , *setegid*() , *seteuid(), setgid*() , *setregid*() , *setreuid*() , *setuid*()
- **u oblast** a **proc záznam**
- datové struktury obsahující řídící informace pro procesy
- často tabulka procesů s položkami **proc** záznam má pevnou velikost v adresovém prostoru jádra
- nověji pole ukazatelů, na dynamicky vytvářené **proc** záznamy, ale pole pevné velikosti, SVR4
- u oblast je mapovaná do uživatelského adresového procesu → je viditelná, když je proces běžící, často mapovaná na pevnou virtuální adresu, proměnná u v jádře
- **`proc`** záznam procesu je přímo přístupný i když proces není běžící

- **`u oblast`**
  1. ukazatel na **`proc`** záznam
  2. reálný a efektivní UID a GID
  3. argumenty a návratové hodnoty systémových volání
  4. ošetření signálů
  5. tabulka deskriptorů otevřených souborů
  6. okamžitý adresář a řídící terminál
  7. využití CPU a kvóty
  8. v mnoha implementacích zásobník jádra

- **`proc záznam`**
  1. identifikaci PID
  2. umístnění u oblasti
  3. stav
  4. ukazatele na vytvoření seznamů všech procesů, čekajících procesů, ...
  5. událost, na kterou proces čeká
  6. informace pro plánování
  7. pole neošetřených signálů
  8. informace pro správu paměti
  9. propojení na PID v rozptýlené (hash) tabulce
  10. informace pro hierarchii procesů

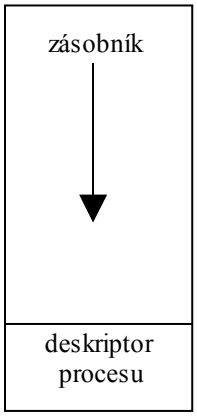
5.4 Implementace procesu - Linux

- jedna údajová struktura – deskriptor procesu, **task_struct**
- položky obsahují ukazatele, např. na informace o prostředcích

`tty_struct`	terminál sdružený s procesem
`fs_struct`	okamžitý adresář
`files_struct`	ukazatele na deskriptory souborů
`mm_struct`	ukazatele na deskriptory oblastí paměti
`signal_struct`	přijaté signály

- deskriptor procesu a zásobník jádra v jedné oblasti paměti o velikosti 8KB

```
union task_union {
     struct task_struct
task;
     unsigned long
stack[1024];
};
```

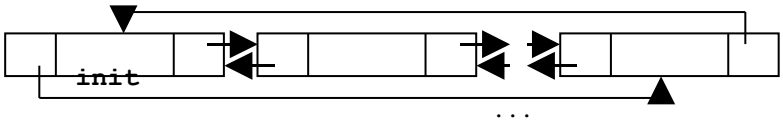


- po přepnutí z uživatelského módu do módu jádro, ukazatel zásobníku ukazuje na vrchol
- maskováním nejnižších 13 bitů (8KB = 2^{13}) získáme adresu deskriptoru procesu

```
movl $0xffffe000, %ecx
andl %esp, %ecx
movl %ecx, p
```

- výhodné pro multiprocesorové systémy
- pro efektivní hledání, např. všech připravených procesů, deskriptory procesů jsou kruhově obousměrně spojeny – kruhový obousměrný spojový seznam
- ukazatelé jsou položky **task_struct**

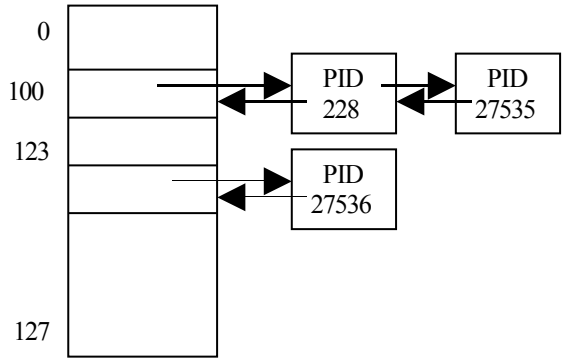
- seznam procesů (*process list*)



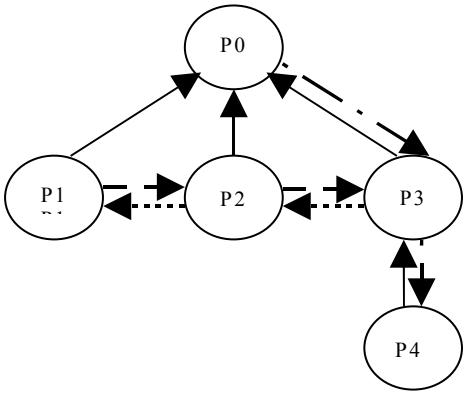
- procesy mohou být ve více seznamech
  - seznam připravených (r*unqueue*)
  - seznamy čekajících na nějakou událost (*wait queues*)
- seznamy umožní nalézt všechny nebo všechny procesy s nějakou vlastností
- jádro někdy musí určit deskriptor procesu z jeho PID
  - zaslání signálu procesu systémovým voláním
    `kill(pid, sig)`
  - řešení
    - projít seznam všech procesů a zjišťovat shodu **pid** a PID v deskriptoru procesu ?
    - vytvořit pole, kterého prvek s indexem **pid** obsahuje ukazatel na deskriptor procesu ?

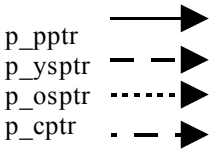
- rozptýlená tabulka (*hash table*)
  - položka obsahuje ukazatel na deskriptor procesů se zadaným PID
  - v případě kolize jsou odpovídající deskriptory zřetězeny
  - ukazatele jsou součástí

```
#define pid_hashfn(x) ((((x) >> 8 ^ (x)) &
(PIDHASH_SZ-1))
```



- hierarchie procesů – vztahy rodič/potomek se vytváří položkami v deskriptoru procesů
  p_opptr originální rodič
  p_pptr rodič, rozdíl např. po ptrace()
  p_cptr nejmladší potomek
  p_ysptr mladší sourozenec
  p_osptr starší sourozenec
  proces P0 vytvořil postupně procesy P1, P2, P3 a proces P3 vytvořil proces P4





6 Výpočet v módu jádro, systémová volání, výjimky, přerušení

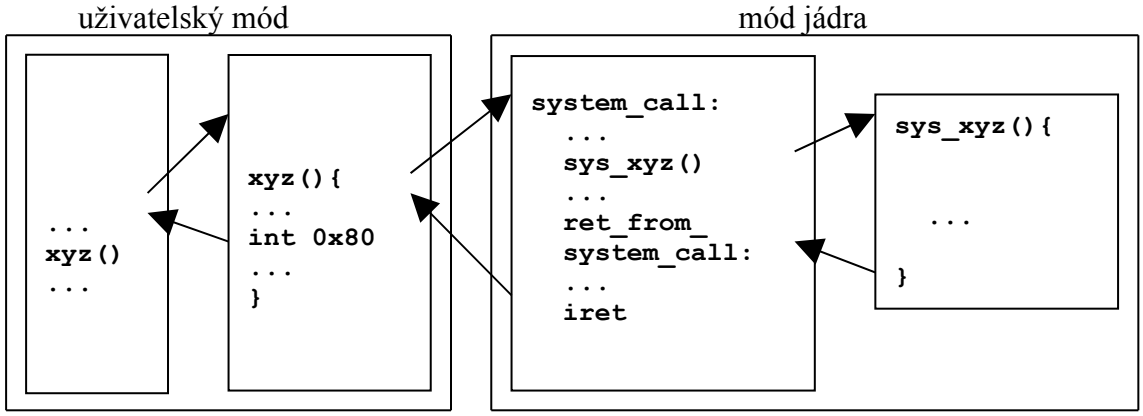
6.1 Výpočet v módu jádro

- v důsledku událostí
  - přerušení (od zařízení asynchronně)
  - výjimky
  - softvérové přerušení
- řízení se předá na proceduru pro ošetření odpovídající události
- část stavu přerušeného procesu potřebná pro obnovení jeho vykonávání po skončení obsloužení události (počítadlo instrukcí, PSW) se uloží do zásobníku jádra přerušeného procesu

6.2 Systémové volání

- v standardní knihovně jazyka C je pro každé systémové volání obálková procedura, řízení se předá softvérovým přerušením proceduře jádra, která se nazývá **`syscall(), system_call,`** protože je jedna pro všechny služby, požadovaná služba je identifikována parametrem procedury, který se nazývá číslo systémového volání

Linux



- aplikační program volá službu **xyz** (např. **fork()**)
- obálková procedura uloží číslo služby do registru **eax** (5 pro **fork()**) před vykonáním **int 0x080**
- **`system_call:`**
  - uloží obsah registrů (HW kontext)
  - zavolá odpovídající funkci (v jazyku C)
  - ukončí se voláním **ret_from_sys_call()**

6.3 Výjimky se spracují obdobně

- jsou synchronní s procesem (vznikají v důsledku událostí způsobených vykonáváním procesu)
- procedury pro jejich zpracování mají obdobnou strukturu jako
- procedura pro systémová volání **`system_call`**

Linux

- Procedura pro zpracování výjimky:
  - uloží obsah registrů
  - zpracuje výjimku (funkce v jazyku C)
    - pošle signál procesu
    - zpracuje žádost o stránku
  - ukončí se voláním funkce **`ret_from_exception()`**

6.4 Zpracování přerušení

- přerušení je obecně asynchronní vzhledem k přerušenému procesu
  - proces čeká na přenos dat, po dokončení přenosu je přerušen úplně jiný proces
- zpracování přerušení nesmí způsobit čekání, přerušený proces zůstává ve stavu běžící
- čas zpracování přerušení je započítán přerušenému procesu, při zpracování přerušení se tedy přistupuje do jeho záznamu **proc**
- obsloužení přerušení:
  - uloží IRQ (*Interrupt ReQuest)* a obsah registrů
  - pošle potvrzení PIC (*Programmable Interrupt Controller)*
  - vykoná obslužní proceduru přerušení
  - ukončí se skokem na **`ret_from_intr()`**

6.5 Vzájemné vnoření systémového volání, výjimek a přerušení

Předpokládejme odladěné jádro

- zpracování systémového volání
  - může vzniknout výjimka žádost o stránku (výpadek stránky)
  - může vzniknout přerušení
- zpracování výjimky (jakékoliv)
  - může vzniknout přerušení
- zpracování přerušení
  - může vzniknout přerušení

při každém odkladu zpracování některé z uvedených událostí musíme uložit odpovídající HW kontext

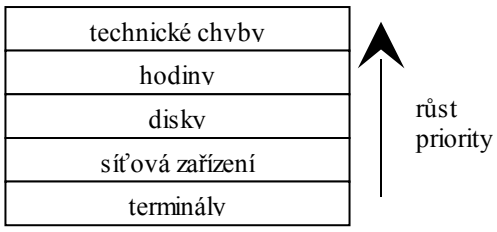
- vytváří se kontextové vrstvy v zásobníku jádra přerušeného procesu
- existuje globální zásobník přerušení

přerušení nejsou všechna stejně naléhavá

- prioritní schéma
- odložení vykonání nekritických akcí obsluhy

6.6 prioritní schéma

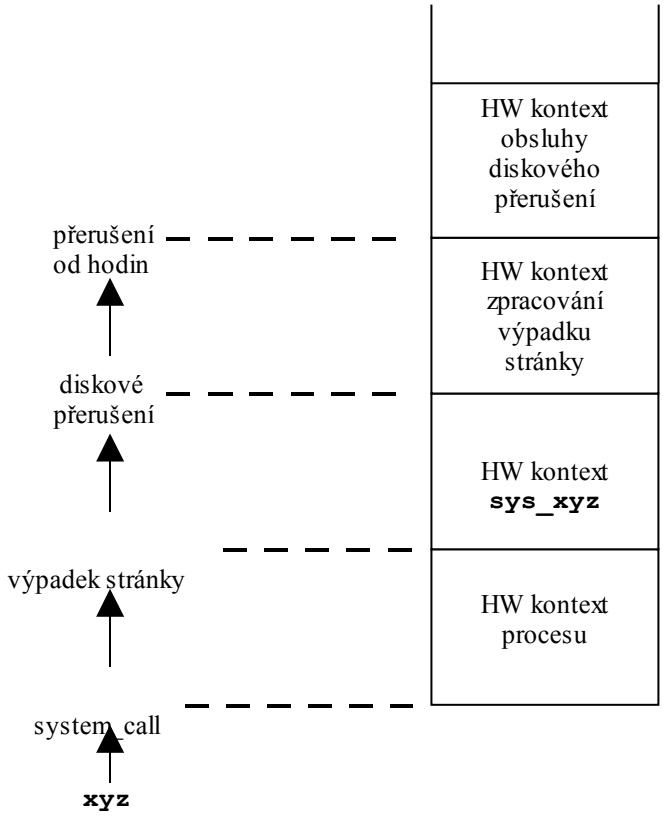
- přerušení mají přiřazené prioritní úrovně (*interrupt priority level*)



- ve stavovém registru procesoru je nastavena okamžitá prioritní úroveň zpracovávaného přerušení

- vznikne-li přerušení s nižší nebo stejnou prioritní úrovní, je uloženo a jeho obsluha je odložena
- vznikne-li přerušení s vyšší prioritní úrovní, uloží se HW kontext, na tuto vyšší prioritní úroveň se nastaví hodnota okamžitého přerušení ve stavovém registru procesoru, zpracuje se přerušení
- při skončení zpracování se z uloženého PSW obnoví okamžitá prioritní úroveň přerušení

Příklad:
aplikační program → **`xyz`** → výpadek stránky → přerušení od terminálu → diskové přerušení → přerušení od hodin



6.7 odložení vykonání nekritických částí obsluhy přerušení (Linux)

- akce při obsluze přerušení se dělí do tří tříd
- kritické, potvrzení přerušení, aktualizace dat používaných zařízením i procesorem, přerušení jsou zakázána (*disabled*)
- nekritické, data používána jenom procesorem, přerušení uvolněna (*enabled*)

- nekritické odložitelné, kopírování vyrovnávací paměti do adresového prostoru procesu, vykonávané funkcí jádra *bottom half*, vykonají se však před **`ret_from_intr()`**

- umožňuje prokládání (*interleaving*) vykonávání obsluh přerušení
- zlepšuje propustnost PIC a řadičů zařízení, nemusí čekat na dokončení obsluhy předcházejícího přerušení
- zjednodušuje kód jádra a zlepšuje přenositelnost

6.8 Vytvoření procesu – fork()

1. Přiděl nový PID a **proc** záznam.
2. Inicializuj **proc** záznam potomka.
   - UID a GID, maska signálů, ... se kopírují z rodiče
   - čas využití CPU, ... se nulují
   - PID, PID rodiče se nastaví pro potomka.
3. Zvyš počet odkazů na i-uzel okamžitého adresáře a
4. případně na i-uzel změněného kořenového adresáře.
5. Zvyš počet odkazů na otevřené soubory v tabulce souborů.
6. Přiděl potomkovi tabulku oblastí.
7. Přiděl potomkovi u oblast a zkopíruj ji z rodiče.
8. Přidej potomka k procesům sdílejícím oblast textu (kódu), který vykonává rodič.
9. Zdvoj oblast dat a zásobníku.
10. Inicializuj HW kontext potomka kopírováním registrů rodiče.
11. Nastav stav na **připraven na vykonání** a vlož ho do fronty pro plánovač.
12. Potomkovi vrať hodnotu 0.
13. Rodiči vrať hodnotu PID potomka.

6.9 optimalizace

- **fork** vždycky vytvářel nový adresový prostor pro potomka
- *copy-on-write* (kopíruj při zápisu), System V a další
  - potomek dostane vlastní kopii tabulky oblastí
  - oblast dat a zásobníku (jejich stránky) jsou dočasně *read-only* (přístup jenom pro čtení) a označené jako *copy-on-write*
  - pokusí-li se rodič nebo potomek modifikovat stránku těchto oblastí vznikne výjimka
  - výjimka se zpracuje tak, že se vytvoří zapisovatelná kopie stránky
  - zavolá-li potomek **exec** nebo **exit**, stránkám rodiče se vrátí jejich původní ochrana
- systémové volání **vfork()**, BSD
  - jestli očekáváme po **fork** brzké volání **exec** můžeme použít nové systémové volání **vfork**()
  - potomek je vykonáván v původním adresovém prostoru rodiče, který spí do jejich vrácení
  - když potomek vykoná **exec** nebo **exit** jádro adresový prostor rodiči a vzbudí ho

6.10 Vyvolání programu – exec()

- program je ve vykonatelném souboru – **a.out, coff, elf**

1. Analyzuj cestu k souboru a zpřístupni soubor, i-uzel.
2. Ověř, že volající má oprávnění na jeho na jeho vykonání.
3. Přečti hlavičku souboru a ověř, že soubor je vykonatelný.

4. Má-li soubor nastaven bit SUID nebo SGID, změň efektivní UID a uložený nastavovací UID na UID vlastníka souboru.
5. Zkopíruj argumenty a proměnné okolí do adresového prostoru jádra.
6. Uvolni paměťové oblasti procesu. Byl-li proces vytvořen službou **vfork**, vrať adresový prostor rodiči.
7. Vytvoř nový adresový prostor. Je-li oblast textu již aktivní bude sdílená, jinak se inicializuje ze souboru.
8. Zkopíruj argumenty a proměnné okolí zpátky do uživatelského zásobníku.
9. Inicializuj HW kontext. Počítadlo instrukcí je nastaveno na adresu vstupního bodu programu.

6.11 Ukončení procesu - exit()

1. Ignoruj všechny signály.
2. Zavři všechny otevřené soubory.
3. Uvolni okamžitý adresář
4. Uvolni, je-li změněný kořenový adresář.
5. Uvolni přidělené paměťové oblasti.
6. Zapiš statistiky o využívání prostředků a parametr **stav** do záznamu **proc.**
7. Změň stav na **mátoha**.
8. PID rodiče všech potomků nastav na 1 – proces **init**
9. Pošli signál skončení potomka SIGCHLD rodiči.
10. Je-li rodič spící, vzbuď ho.
11. Vykonej přepnutí kontextu pro nový naplánovaný proces **swtch()**.

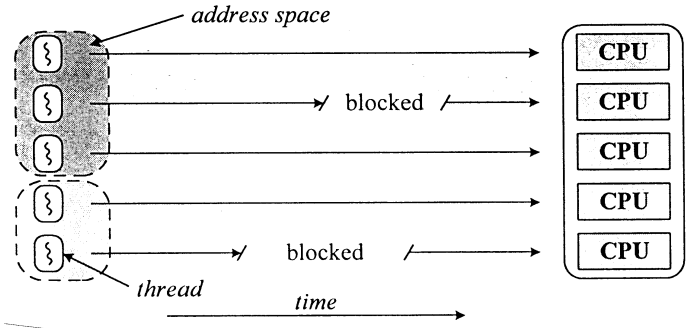
- po skončení volání **exit()** je proces ve stavu **mátoha** a obsazuje záznam **proc**

6.12 Očekávaní skončení potomka - wait()

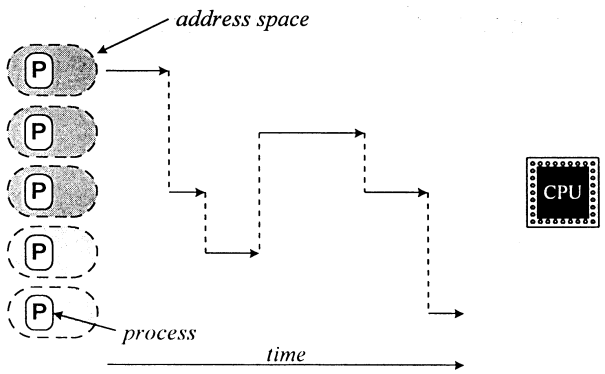
- Chyba, nemá-li proces potomky.
- Má-li potomka mátohu, vyber jednoho z nich, připočítej využití prostředků rodiči, uvolni záznam **proc** vrať PID potomka a stav skončení.
- Jinak přejdi do stavu spící do příchodu signálu SIGCHLD.

7 Vlákna, využití a implementace

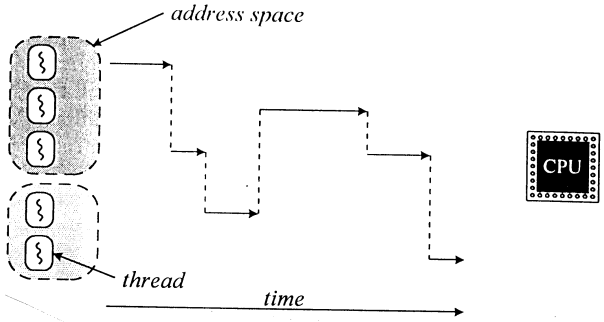
- Mnoho aplikací se skládá z vykonání několika úkolů, které se nemusí vykonávat po sobě v určeném pořadí, ale pracují se společnými prostředky.
- Tomu neodpovídá sekvenční program.
- V tradičních systémech UNIX, takové aplikace používají více procesů.
- Server aplikace
  1. proces přijímač čeká na požadavek klienta
  2. po příchodu požadavky vykoná **fork** a vytvoří proces pro jeho obsluhu
  3. na multiprocesorových systémech paralelní vykonávání
  4. na jednoprocesorových systémech, zlepšení jestliže obsluha požadavku vyžaduje V/V operaci ve srovnání se sekvenčním programem
- Matematické výpočty – výpočet různých maticových operací
  1. výpočty mohou být na sobě nezávislé
  2. můžeme vytvořit proces pro výpočet každého prvku
  3. na multiprocesorových systémech paralelní vykonání
  4. na jednoprocesorových systémech zlepšení např. když při výpočtu nějaké nezávislé části nastane výpadek stránky
- Použití více procesů v jedné aplikaci přináší vysokou režii na vytváření procesů a jejich správu.
- Protože každý proces má svůj vlastní adresový prostor musí se použít prostředky meziprocesové komunikace – roury a sdílená paměť.
- **vlákno** je relativně nezávislá část aplikace vykonávaná sekvenčně, která má jeden tok řízení a může být plánována operačním systémem, existuje v procesu a může používat jeho prostředky
- **proces** může mít jedno nebo více vláken, vykonávaných v tomtéž adresovém prostoru a sdílejících prostředky procesu, např. soubory
- každé vlákno má vlastní zásobník, počítadlo instrukcí a HW kontext
  - efektivní řešení v porovnání s procesy, vyžaduje však synchronizaci na úrovni vláken
  - tradiční procesy v sytému UNIX byly jednovláknové, celý výpočet byl sekvenční



Multithreaded processes on a multiprocessor.



Traditional UNIX system—uniprocessor with single-threaded proces



Multithreaded processes in a uniprocessor system.

7.1 jádrová vlákna (kernel threads)

1. jsou vytvářena a rušena uvnitř jádra pro určené funkce
2. sdílí prostor jádra a má vlastní zásobník
3. je nezávisle plánováno standardními mechanismy **sleep(), wakeup()**
4. není sdruženo se žádným uživatelským procesem
5. efektivní vytváření a používání

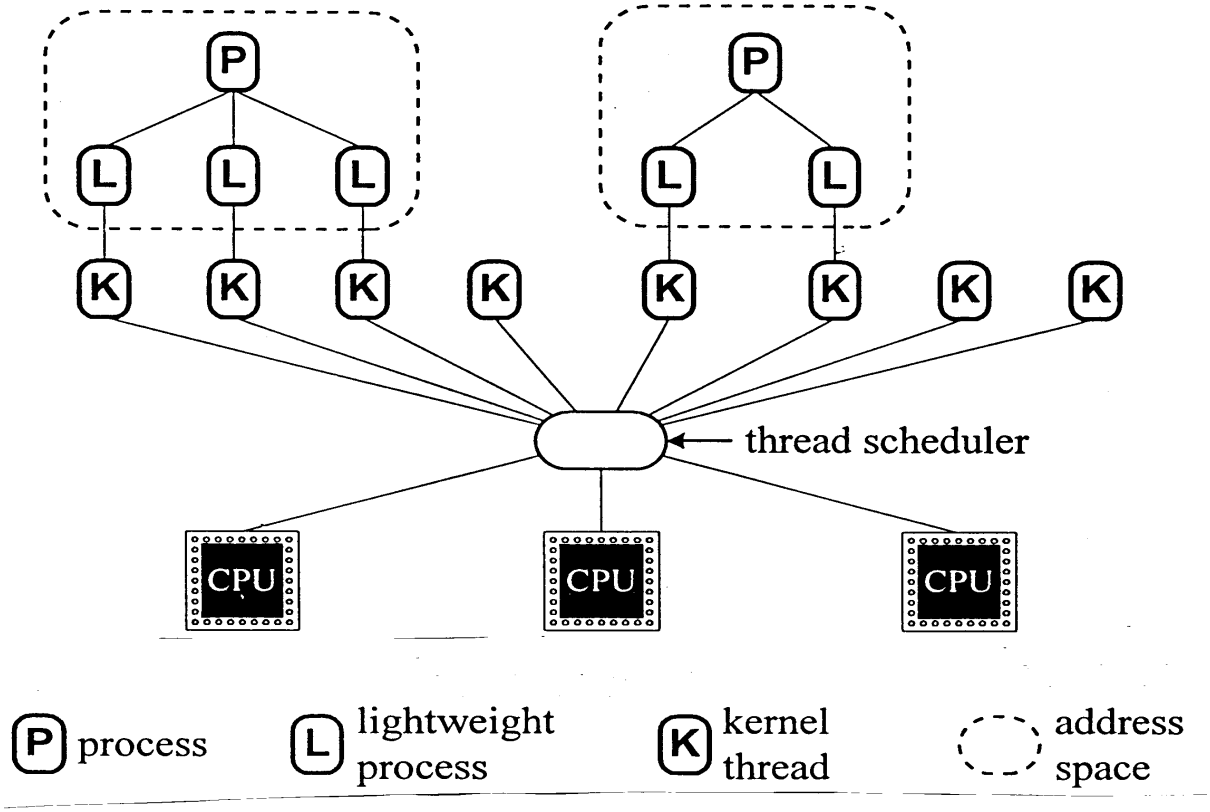
- použití
  1. asynchronní V/V operace
  2. ošetření přerušení
  3. práce s vyrovnávacími paměťmi disků
  4. práce se stránkami paměti
  5. síťové spojení

- koncepčně shodné s démony, které nejsou sdruženy s uživatelským procesem a vykonávají systémové úkoly
- vlákna umožňují jednodušší implementaci

7.2 lehké procesy

1. lehký proces je uživatelské vlákno podporováno jádrovým vláknem
2. proces může mít jeden nebo více lehkých procesů, každý podporován zvláštním jádrovým vláknem
3. lehké procesy jsou nezávisle plánovány a sdílejí adresový prostor a ostatní prostředky procesu
4. můžou vykonávat systémová volání a čekat na prostředky
5. každý lehký proces může být vykonáván na jiném procesoru

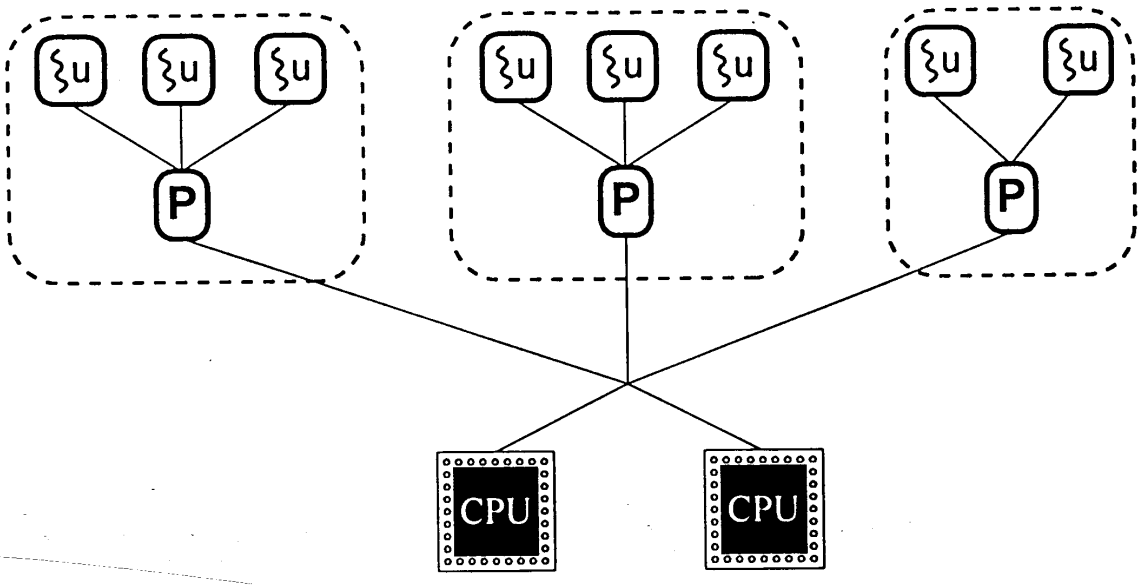
6. přístup k prostředkům více lehkými procesy musí být synchronizován



- omezení
  1. většinu operací s lehkými procesy vykonává jádro
  2. počet lehkých procesů je limitován prostředky OS
  3. lehké procesy musí být dostatečně obecné, aby univerzálně vyhovovali aplikacím
  4. lehké procesy jsou plánovány jádrem

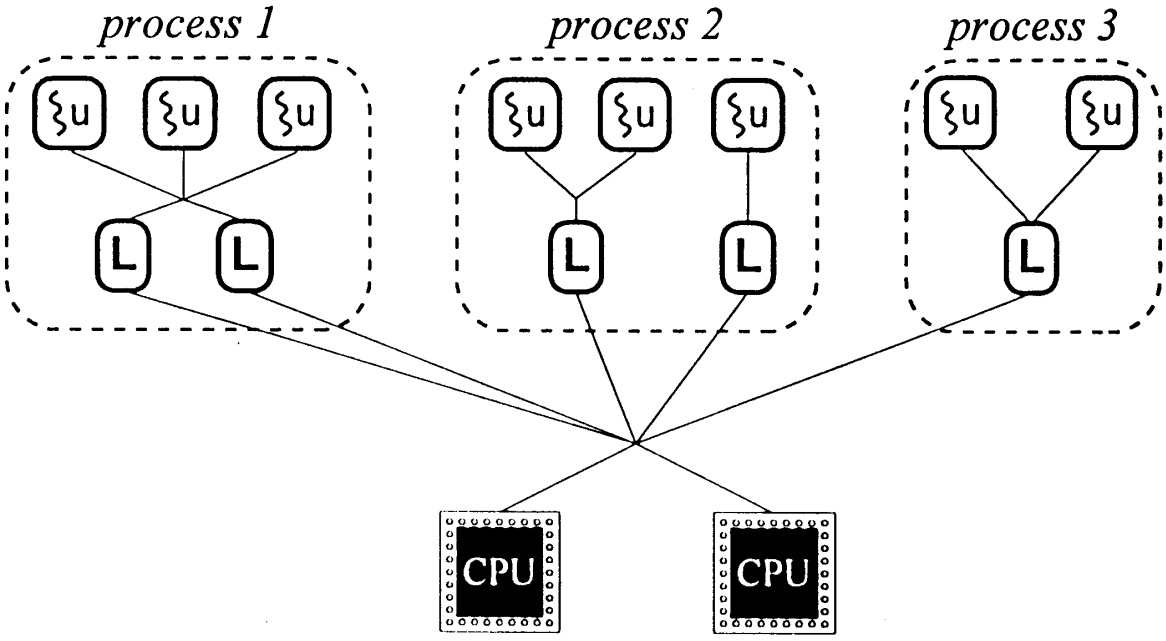
7.3 uživatelská vlákna

- vlákna možno poskytnout na uživatelské úrovni, bez toho aby o nich jádro vědělo
- dosahuje se toho knihovními funkcemi
  - IEEE POSIX 1003.1c
  - Pthreads
  - GNU Pth – The GNU Portable Threads
- interakce vláken nezahrnují jádro a jsou proto velice rychlé
- uživatelská vlákna nemůžou využít paralelizmus multiprocesorových systémů



User threads on top of ordinary processes

- alternativně možno uživatelská vlákna přepínat nad lehkými procesy
  1. jádro vidí, spravuje a přepíná lehké procesy
  2. knihovna vykonává přepínaní uživatelských vláken nad lehkými procesy, zabezpečuje jejich synchronizaci bez vědomí jádra



User threads multiplexed on lightweight libraries

- výhody:
  1. přirozené programování, např. oknové systémy
  2. synchronní model programování
  3. můžeme poskytovat různé knihovny vláken vhodné pro různé aplikace, www.gnu.org/software/pth
  4. výkonnost - časy operací pro uživatelské vlákno, lehký proces a proces (v mikrosekundách)

	čas vytvoření	synchronizace semaforem

uživatelské vlákno	52	66
lehký proces	350	390
proces	1700	200

- nevýhody:

1. jeden adresový prostor a společné prostředky vyžadují synchronizaci na úrovni vláken (procesy využívají oprávnění)
2. plánování lehkých procesů vykonává jádro a nevidí jaké a kolik uživatelských vláken je na nich přepínáno
    1. může vykonat preempci lehkého procesu, na kterém běží proces vlastnící prostředek, který bude požadovat vlákno na novém lehkém procesu
    2. může vykonat preempci lehkého procesu s vláknem s vysokou prioritou
    3. uživatelská vlákna přepínána na jednom lehkém procesu nemůžou být vykonávána paralelně ani na multiprocesorových systémech

7.3.1 Solaris, SVR4.2/MP

1. jádrová vlákna
2. lehké procesy
3. uživatelská vlákna

7.4 jádrová vlákna

1. nezávisle plánována na procesory
2. v jednom adresovém prostoru → efektivnější přepínání mezi vlákny než mezi procesy
3. zásobník
4. údajová struktura
    1. uložení registrů jádra
    2. priorita a plánovací informace
    3. ukazatele pro zařazení do front
    4. ukazatel na zásobník
    5. ukazatele na sdružené záznamy **lwp** a **proc** (NULL, není-li sdružen s lehkým procesem)
    6. ukazatele na udržování fronty všech vláken procesu a všech vláken v systému
    7. informace o sdruženém lehkém procesu

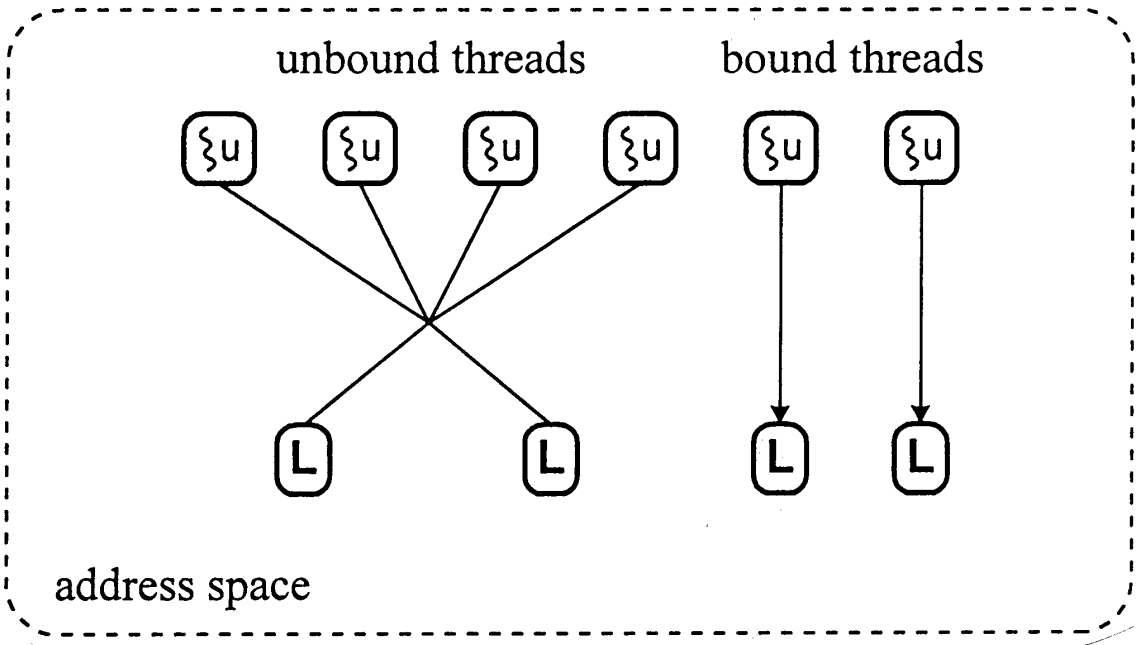
- některé jádrová vlákna vykonávají lehké procesy, jiná vnitřní funkce jádra, zápis na disk

7.5 lehké procesy

- více lehkých procesů může být vykonáváno v jednom procesu
- změna tradičních údajových struktur, u oblasti a **proc** záznamu
- **proc** záznam obsahuje všechny údaje o procesu
- **lwp** záznam obsahuje údaje o lehkém procesu
    1. uložené hodnoty registrů uživatelské úrovně
    2. argumenty systémových volání, výsledky, chybový kód
    3. informaci pro zpracování signálů
    4. využití prostředků
    5. alarmy
    6. využití CPU
    7. ukazatel na jádrové vlákno
    8. ukazatel na **proc** záznam

7.6 uživatelská vlákna

- implementována knihovnou
    1. jsou tvořena, rušena a spravována bez jádra
    2. knihovna poskytuje synchronizační a plánovací prostředky
    3. knihovna vytvoří bank lehkých procesů, na kterých přepíná uživatelská vlákna, M:N model
    4. uživatelskému vláknu může být vyhrazen lehký proces



The process abstraction in Solaris

7.7 implementace:

1. identifikátor vlákna
2. úschova registrů, počítadlo instrukcí a ukazatel na zásobník
3. uživatelský zásobník
4. maska signálů
5. priorita
6. lokální paměť, **errno** nemůže být jedno pro proces

7.8 Linux

- poskytuje knihovní systémové volání **__clone()**
- umožňuje definovat funkci vykonávanou ve vláknu, které části kontextu procesu se zdvojí, a které zůstanou společné
- z pohledu jádra vytváří nový proces - potomka, který však může sdílet s rodičem různé části kontextu
- některé publikace potom procesy, které nemají zdvojeny všechny části kontextu, nazývají lehké procesy (i když nemají „pod sebou“ jádrový proces)
- jádro s nimi zachází stejně jako s procesy
- z pohledu naší klasifikace *jádrová vlákna, lehké procesy, uživatelské procesy* jde o jádrová vlákna
- termín vlákno potom používají pro uživatelská vlákna

- termín jádrová vlákna používají v užším smyslu pro ta jádrová vlákna, které vykonávají systémové úkoly
- **__clone()** má čtyři parametry
- **fn** specifikuje funkci, kterou má potomek vykonat, pojejím vykonání potomek končí
- **arg** ukazatel na argumenty funkce **fn()**
- **flags** obsahuje signál, který se pošle rodiči po skončení potomka a příznaků klonování, které specifikují části kontextu (prostředky) sdílené potomkem a rodičem když jsou nastavené, jsou sdílené
  - **`CLONE_VM`** - tabulka oblastí paměti procesu a všechny stránky
  - **`CLONE_FS`** - kořenový adresář a okamžitý pracovní adresář
  - **`CLONE_FILES`** - tabulku deskriptorů souborů
  - **`CLONE_SIGHAND`** - tabulka zpracování signálů
  - **`CLONE_PID`** - PID (jenom proces 0)
  - **`CLONE_PTRACE`** - je-li rodič sledován voláním **`ptrace()`**, potomek je také sledován
  - **`CLONE_VFORK`** - použito pro **`vfork()`**
- **child_stack** specifikuje ukazatel na uživatelský zásobník potomka
- **__clone()** využívá volání nízko úrovňové služby **sys_clone()**, která má jenom dva parametry: **flags** a **child_stack**
- po návratu **clone()**určí je-li v rodiči nebo potomkovi a potomek vykoná **fn(arg)**

- **fork()**
  - Linux implementuje jako **sys_clone()**se specifikováním signálu **SIGCHLD** a vynulováním příznaků klonování v prvním parametru a hodnotou druhého parametru 0.

- **vfork()**
  - Linux implementuje jako **sys_clone()**se specifikováním signálu **SIGCHLD** a nastavením příznaků **CLONE_VM** a **CLONE_VFORK** v prvním parametru a hodnotou druhého parametru 0.

7.8.1 Využití jádrových vláken Linuxu

1. vykonávání systémových funkcí jádra
2. implementaci uživatelských vláken

7.8.2 Vykonávání systémových funkcí

- vytvoří se jádrové vlákno pro požadovanou funkci
- pseudokód:

```
int kernel_thread(int (*fn)(void *), void * arg,
                  unsigned long flags){
     pid_t p;
     p = sys_clone(flags|CLONE_VM, 0 );
     if (p)    /* rodic */
          return p;
     else {         /* potomek */
          fn(arg);
          exit;
     }
}
```

- **init** proces potom proces 0 vytvoří:

```
kernel_thread(init, NULL, CLONE_FS|CLONE_FILES|
CLONE_SIGHAND);
```

7.8.3 implementace uživatelských vláken

- knihovny implementující **POSIX 1003.1c - Pthreads**

7.8.3.1 vytvoření vlákna

```
pthread_create (thread_id, attr, thread_function,
arg)

/* Initialize the thread descriptor */
  new_thread->p_nextwaiting = NULL;
  new_thread->p_spinlock = 0;
  new_thread->p_signal = 0;
  new_thread->p_signal_jmp = NULL;
  new_thread->p_cancel_jmp = NULL;
  new_thread->p_terminated = 0;
  new_thread->p_detached = attr == NULL ? 0 :
                                attr->detachstate;
  new_thread->p_exited = 0;
  new_thread->p_retval = NULL;
  new_thread->p_joining = NULL;
  new_thread->p_cleanup = NULL;
  new_thread->p_cancelstate = PTHREAD_CANCEL_ENABLE;
  new_thread->p_canceltype = PTHREAD_CANCEL_DEFERRED;
  new_thread->p_canceled = 0;
  new_thread->p_errno = 0;
  new_thread->p_h_errno = 0;
  new_thread->p_initial_fn = start_routine;
  new_thread->p_initial_fn_arg = arg;
  new_thread->p_initial_mask = mask;
  for (i = 0; i < PTHREAD_KEYS_MAX; i++)
                    new_thread->p_specific[i] = NULL;
  /* Do the cloning */
  pid = __clone(pthread_start_thread, new_thread,
          (CLONE_VM | CLONE_FS | CLONE_FILES
           | CLONE_SIGHAND | PTHREAD_SIG_RESTART),
          new_thread);
```

- jako pro procesy, i pro vlákna je deskriptor vlákna a zásobník jedna datová struktura
- funkce **pthread_start_thread** inicializuje vlákno a vykoná **fn(arg)**

```
static int pthread_start_thread(void *arg)
{
  pthread_t self = (pthread_t) arg;
  void * outcome;
  /* Initialize special thread_self processing, if
     any.  */
#ifdef INIT_THREAD_SELF
  INIT_THREAD_SELF(self);
#endif
```

```
  /* Make sure our pid field is initialized, just in
     case we get there before our father has
     initialized it. */
  self->p_pid = getpid();
  /* Initial signal mask is that of the creating
     thread. (Otherwise, we'd just inherit the mask
     of the thread manager.) */
  sigprocmask(SIG_SETMASK, &self->p_initial_mask,
              NULL);
  /* Run the thread code */
  outcome =
          self->p_initial_fn(self->p_initial_fn_arg);
  /* Exit with the given return value */
  pthread_exit(outcome);
  return 0;
}
```

7.8.3.2 ukončení vlákna

```
pthread_exit(stav)
```

7.8.3.3 čekání na skončení vlákna

```
pthread_join (thread_id, stav)
```

- volající vlákno, čeká na skončení vlákna **thread_id**

7.8.4 vzájemné vylučování (*mutual exclusion*) – mutex

- synchronizace přístupu k prostředkům (datům)
- inicializace

```
pthread_mutex_init (mutex, attr)
```

- **mutex** je inicializován jako „odemknutý“ (*unlocked*)

```
pthread_mutex_lock (mutex)
```

- vlákno „zamkne“ (*locks*) **mutex** a pokračuje,
- je-li **mutex** zamknut, vlákno je do odemknutí blokováno

```
pthread_mutex_unlock (mutex)
```

- vlákno „odemkne“ (*unlocks*) **mutex**

7.8.5 podmínkové proměnné (*condition variables*)

- synchronizace při dosažení nějakých hodnot proměnných
- reprezentuje nějakou podmínku, např. **pocet=limit**
- pro přístup k proměnným vyjadřujícím podmínku, musí být s každou podmínkovou proměnnou sdružený mutex
- inicializace

```
pthread_cond_init (condition, attr)
```

- čekání na signalizaci (*signalling*) podmínky

```
pthread_cond_wait (condition, mutex)
```

- pseudokód:

```
pthread_cond_wait (condition, mutex)
begin
     pthread_mutex_unlock (mutex);
     čekej_na_podmínku (condition);
```

```
            pthread_mutex_lock (mutex);
        end
```

- signalizování podmínky

```
    pthread_cond_signal (condition)
```

- vlákno čekající na podmínku je vzbuzen a dokončí **pthread_cond_wait()**, signalizující vlákno musí odemknout sdružený **mutex**
- obecně před úspěšným zamknutím **mutex-**u čekajícím vláknem, mohlo **mutex** zamknout jiné vlákno a splnění podmínky se musí znovu zkontrolovat

7.9 1996 - LinuxThreads

- model 1:1, jedno uživatelské vlákno je podporováno jedním jádrovým vláknem, jednoduchá implementace, možnost využití multiprocesorů pro paralelní vykonávání, není dvojité plánování

7.10 1999 - GNU Portable Threads GNU Pth

- přenositelné pro systémy UNIX, implementuje POSIX vlákna v uživatelském prostoru v rámci procesu

7.11 2002 -

- standardní distribuce POSIX thread knihovny jsou LinuxThreads
- NPTL, Native POSIX Thread Library, zachovává model 1:1
- NGPT, Next Generation POSIX Threading, vychází z GNU Pth, model M:N

8 Signály

- signály umožňují oznámit procesům výskyt událostí v systému
- jde o krátké zprávy, kde se procesům oznámí číslo signálu
- systémová volání pro signály i vnitřní implementace se u jednotlivých variant a verzí značně liší, System V vs. BSD
- problémy:
  - pro tvůrce přenositelných aplikací – může používat taková volání která jsou všude stejná
  - pro výrobce operačních systémů, které chtějí být kompatibilní s více variantami – musí poskytovat všechna systémová volání
- standardní rozhraní specifikuje POSIX, včetně zpětné kompatibility
- čísla některých signálů závisí na HW, označují se symbolickými konstantami SIG...
- signály slouží dvěma hlavním účelům
  - uvědomit proces, že nastala určitá událost
  - přinutit proces vykonat funkci na zpracování signálu (*signal handler*)
- systémová volání umožňují programátorovi zasílat signály a určit jak budou použity

8.1 POSIX

Signal	Default Action	Description
SIGABRT	**A**	**Process abort signal.**
SIGALRM	**T**	**Alarm clock.**
SIGBUS	**A**	**Access to an undefined portion of a memory object.**
SIGCHLD	**I**	**Child process terminated, stopped, or continued.**
SIGCONT	**C**	**Continue executing, if stopped.**
SIGFPE	**A**	**Erroneous arithmetic operation.**
SIGHUP	**T**	**Hangup.**
SIGILL	**A**	**Illegal instruction.**
SIGINT	**T**	**Terminal interrupt signal.**
SIGKILL	**T**	**Kill (cannot be caught or ignored).**
SIGPIPE	**T**	**Write on a pipe with no one to read it.**
SIGQUIT	**A**	**Terminal quit signal.**
SIGSEGV	**A**	**Invalid memory reference.**
SIGSTOP	**S**	**Stop executing (cannot be caught or ignored).**
SIGTERM	**T**	**Termination signal.**
SIGTSTP	**S**	**Terminal stop signal.**
SIGTTIN	**S**	**Background process attempting read.**
SIGTTOU	**S**	**Background process attempting write.**
SIGUSR1	**T**	**User-defined signal 1.**
SIGUSR2	**T**	**User-defined signal 2.**
SIGPOLL	**T**	**Pollable event.**
SIGPROF	**T**	**Profiling timer expired.**
SIGSYS	**A**	**Bad system call.**
SIGTRAP	**A**	**Trace/breakpoint trap.**
SIGURG	**I**	**High bandwidth data is available at a socket.**
SIGVTALRM	**T**	**Virtual timer expired.**
SIGXCPU	**A**	**CPU time limit exceeded.**
SIGXFSZ	**A**	**File size limit exceeded.**

- některé signály jsou vzhledem k procesu asynchronní
  - SIGINT, přerušení od terminálu stisknutím CTRL-C

- jiné jsou synchronní
  - SIGSEV, chyba odkazu na stránku
- jádro při zasílaní signálů rozlišuje dvě fáze:

1. odeslání signálu - jádro zaznamená v záznamu **proc** (deskriptoru procesu) zasílanému procesu odeslání nového signálu
2. přijetí signálu - jádro přinutí proces reagovat na signál

- pro každý signál je nastavená implicitní reakce, která se vykoná, pokud ji proces nespecifikuje jinak
  - T (*abnormal*) *termination,* také *abort, xit*
    proces je násilně ukončen se všemi důsledky volání **exit(stav)**, přitom **stav** indikuje pro **wait()** a **waitpid()** abnormální ukončení
  - A *abnormal termination,* také *dump, abort*
    navíc se vykoná nějaká akce, typicky výpis obsahu paměti procesu a hodnot registrů do souboru s názvem **core**
  - I *ignore*, signál je ignorovaný
  - S *stop*, proces je zastaven (*stopped*)
  - C *continue*, byl-li proces zastaven, může pokračovat, je převeden do stavu připraven, jinak je signál ignorován
- proces může potlačit nastavenou akci a specifikovat jinou akci
  1. explicitně ignorovat signál
  2. zachytit signál a vykonat uživatelem definovanou funkci, která se nazývá, ošetření/obsluha signálu (*signal handler*)
- na druhé straně, proces může obnovit reakci na signál na nastavenou implicitní akci
- proces může signál blokovat, co znamená, že signál nebude přijat dokud signál není odblokován
- signály SIGKILL a SIGSTOP nemůžou uživatelé ignorovat, blokovat nebo specifikovat pro ně obsluhu
- signál je nevyřízen (*pending*), byl-li odeslán, ale nebyl přijat
- jenom jeden signál každého typu může být nevyřízen
- reakci na signál vykonává proces, kterému je signál zaslán, včetně ukončení procesu, to znamená, že musí být aspoň plánován stát se běžícím
- má-li nízkou prioritu může mezi odesláním signálu a jeho přijetím, kdy se vykoná odpovídající akce uplynout dosti dlouhá doba
- další prodlení může způsobit je-li proces v čase odeslání signálu ve stavu
  1. zastaven
  2. spící
- signály pro zastavení procesu (*stop signals*) **SIGSTOP, SIGTSTP, SIGTIN, SIGTOUT** mění okamžitě stav procesu na **zastaven** nebo **spící_a_zastaven** a signál **SIGCONT** je vrací do původního stavu
- co se má stát, když je odeslán signál spícímu procesu?
- činnost jádra záleží na tom proč proces přešel do stavu **spící**
  1. čeká-li na událost, která zakrátko nastane, např. dokončení diskové V/V operace, je spící v kategorii nepřerušitelný a signál je pouze zaznačen jako nevyřízen
  2. čeká-li na událost, o které nevíme kdy nastane nebo dokonce nemusí nastat vůbec, např. skončení potomka, vstup z terminálu, je spící v kategorii přerušitelný, je jádrem vzbuzen a přejde do stavu připraven
- Linux nemá stav spící, ale stavy **úloha_přerušitelná** (***TASK_INTERRUPTIBLE***) **úloha_nepřerušitelná** (***TASK_UNINTERRUPTIBLE***)

- přijímající proces je přinucen vykonat odpovídající akci, když pro něj jádro zavolá funkci **issig()** na zjištění nevyřízených signálů
- jádro zavolá **issig():**
  1. před návratem do uživatelského módu ze sytémového volání nebo z obsluhy přerušení
  2. před zablokováním procesu v přerušitelné kategorii
  3. když se stane běžícím po vzbuzení ze stavu spící v přerušitelné kategorii
- když proces začal vykonávat systémové volání a nastane některý z posledních dvou případů, proces přijme signál a namísto jeho dokončení vykoná obsluhu signálu a systémové volání se obvykle vrátí s hodnotou **EINTR** v proměnné **errno**

8.1.1 scénář asynchronního signálu

- uživatel stiskne **CTRL-C**
- generuje se přerušení (jako u každého stisknutí klávesy)
- ovladač rozpozná, že jde o kombinaci generující signál a odešle signál **SIGINT** procesu v popředí
- když je proces naplánována jako běžící při návratu do uživatelského módu anebo byl-li běžící při návratu z přerušení proces najde signál

8.1.2 scénář synchronního signálu

- výjimka (dělení nulou, nedovolená instrukce,..) způsobí přechod do módu jádro
- jádro vykoná její obsluhu a zašle se odpovídající signál běžícímu procesu
- při návratu z obsluhy proces najde signál

8.1.3 nespolehlivé signály

- funkce pro obsluhu signálů nejsou perzistentní, po zachycení (nalezení) signálu, jádro ještě před vyvoláním funkce obsluhy signálu nastaví implicitní akci, tedy pro následující signál, chceme-li opět vykonat obslužní funkci musíme ji znovu instalovat, vzniká soutěž (*race condition*)

- instalace obslužní funkce signálu

```
oldfunction=signal(sig, function);
```

- zaslání signálu

```
kill(pid,sig);
```

- Příklad

```
sig_obsluha() {
     printf("signal zachycen");
     signal(SIGINT, sig_obsluha);
}
main() {
     int rpid;
     signal(SIGINT, sig_obsluha);
     if (fork == 0) {
          sleep(5);
          rpid = getpid();
          for(;;)
               if (kill(rpid,SIGINT) == -1)
                    exit(1);
     }
```

```
        /* snížíme prioritu */
        nice(10);
        for(;;);
    }
```

- rodičovský proces má nízkou prioritu a je-li mu odebrán procesor v obslužné funkci signálu **SIGINT** a potomek zašle další signál, proces rodič při jeho přijetí vykoná nastavenou implicitní akci, tj. **exit**
- bylo by řešením nenastavovat implicitní akci?
- ano, ale při obsluze signálu by mohla být vnořena další obsluha, ... a uživatelský zásobník by mohl přetéct

8.1.4 spolehlivé signály

- perzistentní obslužné funkce signálů
- blokování signálu, např. při obsluze signálů → nevznikne hnízdění
- zaslání signálu

```
kill(pid, sig);
pid > 0    signál je zaslám procesu s PID = pid
pid = 0    signál je zaslán všem procesům skupiny
pid = -1   signál je zaslán všem procesům, kromě 0, 1 a běžícího
pid < -1   signál je zaslán všem procesům v skupině –pid
```

- instalace obslužní funkce (náhrada signal)

```
sigaction(sig, act, oact);
```

- specifikuje obsluhu pro signál **sig**
- **act** ukazuje na záznam, který obsahuje:
- akci – **SIG_IGN, SIG_DFL**, nebo obslužní funkci
- masku signálů, které mají být blokovány při vykonávání obslužní funkce
- příznaky
  - **SA_NOCLDSTOP** negeneruj **SIGCHLD**, když je potomek zastaven
  - **SA_RESTART** signálem přerušené systémové volání, se restartuje obslužní funkce musí používat reentrantní systémová volání
  - **SA_ONSTACK** obslož signál v zásobníku deklarovaném voláním sigalstack()
  - **SA_RESETHAND** akce se nastaví na implicitní
  - **SA_SIGINFO** není-li nastaven obslužní funkce je zadána ve tvaru **func (sig);** je-li nastaven obslužní funkce je zadána ve tvaru **func (sig, info, kontext);** kde **info** vysvětluje příčinu vzniku signálu a **kontext** odkazuje na přerušený kontext procesu, když byl signál dodán
  - **SA_NOCLDWAIT** nevytvářej mátohy, když potomci volajícího procesu skončí, zavolá-li proces **wait ()** čeká až všichni potomci skončí
  - **SA_NODEFER** neblokuj automaticky signál, když bude obsluhován

- **oact** volitelně vrátí předcházející akci signálu
- zjištění nevyřízených signálů

```
sigpending(set);
```

- modifikování blokovaných signálů

```
sigprocmask(how, set, oset);
```

- **oset** stará maska signálů
- **set** nová maska signálů
- `how`
  - **SIG_BLOCK** nová maska specifikuje signály, které se přidají k blokovaným

- **SIG_UNBLOCK** nová maska specifikuje signály, kterých blokování se odstraní
- **SIG_SETMASK** nová maska specifikuje blokované signály

- čekání procesu na signál
```
sigsuspend(sigmask);
```
- nastaví se blokované signály podle **sigmask** a proces přejde do stavu čekající až do zaslání signálu, který není blokován nebo ignorován
- není ekvivalentní dvojici **sigprocmask()** a **sleep()**
- systémové volání **sigprocmask()** mohlo odblokovat signál, na který chceme čekat a může se stát, že signál bude přijat před zavoláním sleep() a čekání nemusí skončit

8.1.5 implementace

- základní datová struktura pro uložení odeslaných signálů je pole bitů typu **sigset_t**, jeden bit pro každý signál
```
typedef struct {
     unsigned long sig[2];
} sigset_t;
```
- **0** nemá žádný signál, v prvím prvku 31 tradičních signálů, ve druhém prvku signály pro reálný čas
- deskriptor procesu obsahuje položky
- **signal** typu **sigset_t** označující dodané signály
- **blocked** typu **sigset_t** označující blokované signály
- **sigpending** příznak, který je nastaven je-li jeden nebo více blokovaných signálů nevyřízeno
- **gsig** ukazatel na záznam **signal_struct** opisující obsluhu signálu

```
struct signal_struct {
     atomic_t                count;
     struct k_sigaction  action[64];
     spinlock_t              siglock;
};
```

- **count** počet procesů (a vláken) sdílejících **signal_struct – clone(), fork(), vfork()**
- **siglock** zajišťuje výhradný přístup k položkám **signal_struct**
- **action[64]** 64 **k_sigaction** záznamů specifikujících obsluhu jednotlivých signálů
  - **sa_handler - SIG_IGN, SIG_DFL**, nebo ukazatel na obslužní funkci
  - **sa_flags –** příznaky pro obsluhu signálu
  - **sa_mask –** maskované signály při obsluze

9 Plánování, plánovací třídy, inverze priority

- procesor(y) je sdílen více procesy
- proces má přidělen procesor na časové kvantum (*time quantum*) – sdílení času (*time sharing*)
- pokud neskončí, procesor je přidělen novému procesu – přepnutí procesů
- plánování pojednává o tom, kdy vykonat přepnutí procesů, a který proces vybrat
- plánovací strategie - pravidla na určení kdy vykonat přepnutí procesů, a který proces vybrat
- implementace plánování - algoritmy a datové struktury na implementaci plánovací strategie

9.1 plánovací strategie

- v systémech UNIX/Linux tradičně vychází ze sdílení času
- v systému souběžně může běžet několik aplikací
- odpovídající procesy podle jejich požadavků můžeme rozdělit na:
  1. interaktivní procesy
     čekají na stisk klávesy, kliknutí myší, po vstupu se však čeká rychlá reakce, krátká doba odpovědi (*response time*), průměrně mezi 50-150 ms, ale i s omezeným rozptylem, příkladem je shell, editory, grafické aplikace
  2. dávkové procesy
     nevyžadují bezprostřední interakci s výpočtem, často jsou vykonávány v pozadí, měřítkem efektivnosti plánování je propustnost systému (*troughput*), příkladem jsou kompilátory, vyhledávání v databázích, vědecké výpočty

  1. procesy reálného času
     mají velmi přísné požadavky na plánování, zaručující dobu krátkou dobu odezvy s minimálním rozptylem, například při zpracování video informace může být upřednostněno zobrazování konstantního počtu 15 snímků za sekundu, namísto zobrazování 10 až 30 snímků za sekundu, s vyšším průměrem 20 snímků za sekundu

- alternativní klasifikace tradičně dělí aplikace na:
  1. vědecko-výzkumné výpočty - mají vysoké nároky čas procesoru („*CPU-bound*“)
  2. zpracování hromadných údajů - potřebují množství V/V operací a velkou část času stráví čekáním ne jejich vykonání („*I/O bound*“)
- v systémech UNIX/Linux mají procesy zvláštní třídu(y) plánování (*scheduling class*) pro procesy reálného času a zvláštní třídu(y) plánování pro ostatní procesy
- na druhé straně není nikde specifikováno, které procesy požadují především čas procesoru, a které většinou čekají na dokončení V/V operací
- plánovací strategie je založena na pořadí, ve kterém procesy požadují o přidělení procesoru a na prioritách, které mají přednost
  1. procesy reálného času - statické priority
  2. ostatní - dynamické priority
- procesům, které dlouho nečerpají své časové kvantum prioritu zvýšíme a opačně procesům, které jsou dlouho běžící penalizujeme
- interaktivní aplikace budou mít dobrou dobu odezvy
- volba časového kvanta

1. krátké trvání způsobuje vysokou režii - je-li např. doba přepnutí mezi procesy 10 ms a časové kvantum je rovněž 10 ms, režie je nejmíň 50%
2. příliš dlouhé trvání způsobuje ztrátu zdání souběžného vykonávání procesů - je-li časové kvantum 4 sekundy, a **n** dalších procesů je připraveno, pak bude typicky naplánován za 4***n** sekund
3. dlouhé trvání časového kvanta nemusí způsobit zvýšit dobu odpovědi, procesy které mají interaktivní charakter mají vysokou dynamickou prioritu a rychle vykonají preempci dávkových procesů a to bez ohledu na délku časového kvanta
4. v některých případech však zvýšení doby odpovědi při dlouhém trvání časového kvanta může nastat, například, dva rovnocenní uživatelé **A** a **B** zadají příkazy, přičemž **A** zadal interaktivní a zadal **B** dávkový příkaz, shell pro každého z nich vytvoří proces a předpokládejme u nich po vytvoření stejnou prioritu, operační systém neví který je který a naplánuje-li jako první dávkový příkaz, interaktivní proces bude do začátku jeho vykonávání čekat celé časové kvantum

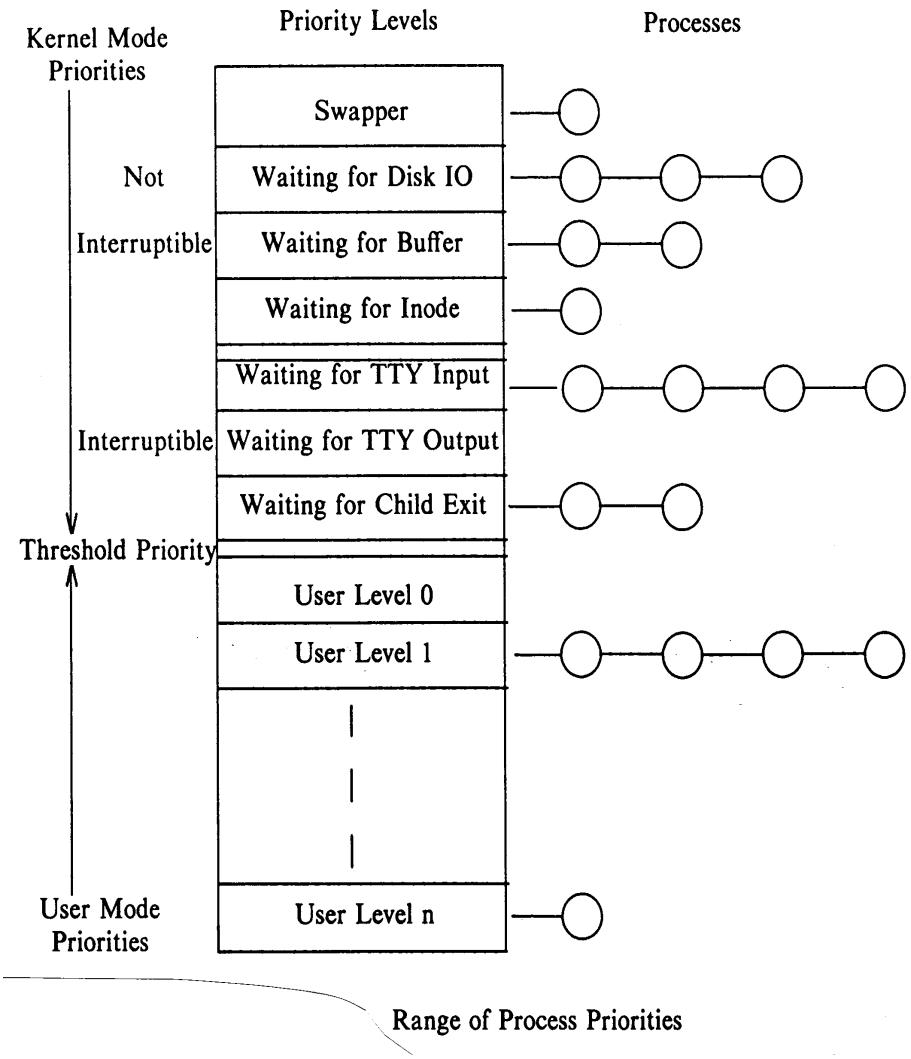
- praktické pravidlo:
  - zvolte trvání časového kvanta co nejdelší pří zachování dobrého času odpovědi systému

9.2 implementace plánování

- technické vybavení má hodiny, časovač (*clock, timer*), který periodicky generuje přerušení, jejich frekvence je typicky 100Hz, tedy tikne každých 10ms
- zpracování přerušení od časovače
  - aktualizuje čas od začátku práce systému
  - aktualizuje čas a datum
  - aktualizuje statistiky o využívání prostředků
  - určuje jak dlouho běžel okamžitý proces
  - kontroluje jestli neuplynul čas sdružený s každým
  - programovým časovačem a když ano vyvolá
  - odpovídající funkci, využíváno jádrem i procesy
- jádro například vypne mechaniku pružného disku, pokud se k němu jistou dobu nepřistupovalo
- uživatel může voláním **setitimer()** a **alarm()** způsobit, aby procesu byly periodicky nebo jednotlivě zasílány signály po uplynutí stanoveného času

9.2.1 tradiční plánování

- vybere se proces ve stavu připraven s nejvyšší prioritou
- má-li nejvyšší prioritu více procesů, vybere se ten, který je připraven nejdéle, cyklické plánování (*round robin*)
- každý proces má ve svém **proc** záznamu položku s prioritou pro plánování, nastavenou při vytvoření procesu
- větší číselná hodnota znamená nižší prioritu
- priority v módu jádro
  1. nepřerušitelné
  2. přerušitelné
- priority v uživatelském módu
  1. procesům, které přecházejí do stavu spící jádro přiradí pevné priority podle příčiny přechodu, priorita nezávisí na charakteru procesu
  2. při přechodu z módu jádra do módu uživatel, jádro upraví prioritu na uživatelskou
- schéma rozsahu priorit (nejvyšší priorita je 0)



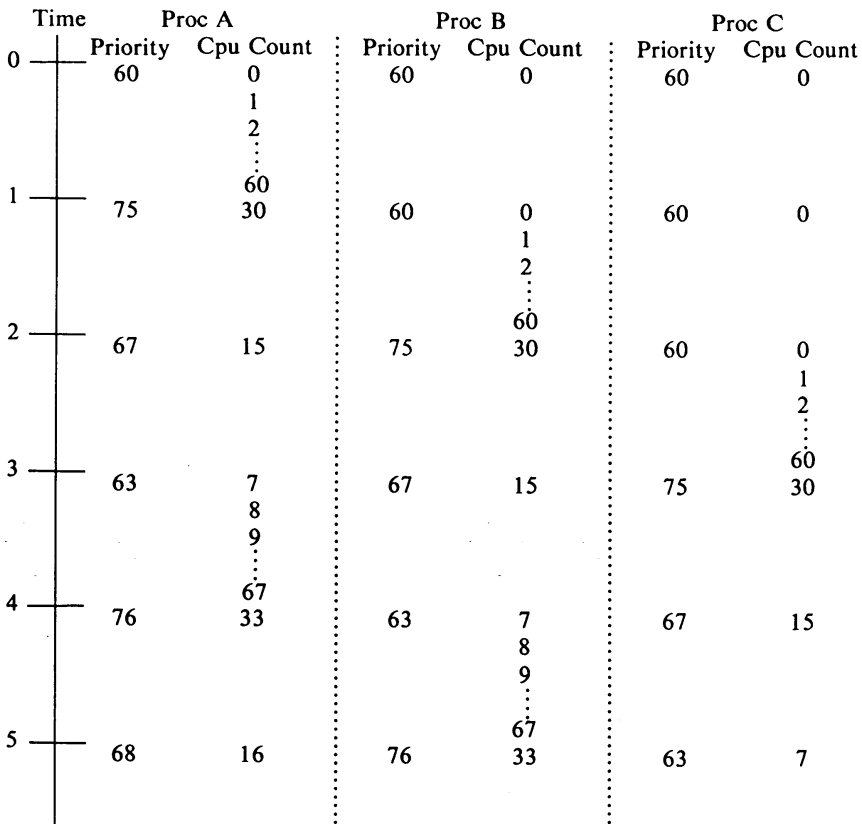
- pro plánování procesů v uživatelském módu jádro pro běžící proces po každém tiknutí časovače zvyšuje hodnotu položky **nedávné_použití_CPU** záznamu **proc**
- jádro každou sekundu (SVR3, 4.3BSD) přepočítá priority všech procesů v uživatelském módu

- sníží hodnotu **nedávné_použití_CPU** funkcí **pokles** (*decay*)
  - **nedávné_použití_CPU = pokles(nedávné_použití_CPU)**
- a prioritu vypočte ze vztahu
  - **priorita = nedávné_použití_CPU/2 + základní_uživatelská_priorita**
- proces, který nedávno čerpal hodně času procesoru, bude mít nízkou prioritu
- funkce **pokles** postupně zabezpečuje růst priority procesům, které čerpali čas procesoru dávno

9.2.2 Příklad

Mějme 3 procesy **A, B, C**, které byly vytvořeny současně se začáteční prioritou 60, nejvyšší uživatelská priorita je 60 a frekvence tiknutí je 60 s^{-1} a jiné procesy nejsou. Funkce **pokles** je

```
pokles(nedávné_použití_CPU) = nedávné_použití_CPU / 2
```



- systémové volání **nice()**
  1. umožňuje změnit prioritu procesu o hodnotu argumentu volání **nice()**

```
priorita = nedávné_použití_CPU/2 +
     základní_uživatelská_priorita +
     hodnota_argumentu_nice
```

  2. jenom privilegovaný uživatel může zadat zápornou hodnotu a tím zvýšit prioritu procesu
  3. neprivilegovaný uživatel, může jenom snížit prioritu svých procesů, je tedy nice k ostatním
- SVR3 nemá plánovací třídu pro procesy reálného času, jádro je nepreemtivní

9.2.3 SVR4

- plánovací třídy: priority:
- reálný čas 100 – 159
- systém 60 – 99
- sdílení času 0 – 59

- vysoká hodnota = vysoká priorita

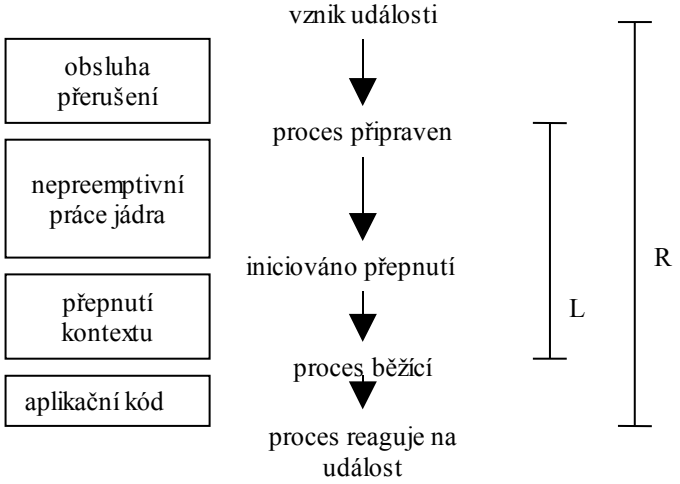
9.2.4 plánovací třída pro sdílení času

1. dynamicky mění priority a pro procesy stejné priority používá cyklické plánování
2. pro každou prioritu je definováno časové kvantum a další parametry
3. pro nízké priority je časové kvantum delší
4. změna priorit pro plánování řízena událostmi v systému, vyčerpání časového kvanta, obvykle přepočet priority pro jeden proces

9.2.5 plánovací třída pro reálný čas

1. statické priority a pevné časové kvantum
2. omezená
   1. plánovací latence (*dispatch latency*)
   2. doba odpovědi (*response time*)

- při vzniku události, na kterou musí proces reálného času reagovat, je běžící proces přerušen a na konci obsluhy odpovídajícího přerušení je proces reálného času převeden do stavu připraven
- pokud byl proces přerušen při vykonávání systémového volání, protože jádro je nepreemtivní může být naplánován při návratu do uživatelského módu



L - plánovací latence

R - doba odpovědi

- pro snížení plánovací latence jádro SVR4 definuje body preempce, jsou to místa v kódu jádra, kdy jsou všechny údaje v konzistentním stavu a jádro se pouští do delšího výpočtu
- po dosažení bodu preempce jádro zjišťuje, jestli není proces reálného času ve stavu **připraven** a je-li tomu tak vykoná preempci běžícího procesu

9.3 Linux

- čas procesoru je rozdělen do epoch (*epoch*)
- v jedné epoše, každý proces má specifikováno časové kvantum vypočteno na jejím začátku
- v jedné epoše proces může využívat své časové kvantum po částech
- epocha končí, když všechny běhu schopné procesy vyčerpali svá časová kvanta, kdy jsou přepočítána časová kvanta všech procesů (ne jenom běhu schopných) a začne nová epocha
- každý proces má základní časové kvantum
- potomek zdědí základní časové kvantum rodiče

- uživatel může změnit základní časové kvantum voláním **nice()** a **setpriority()**
- proces **0** má nastavenou prioritu na **DEF_PRIORITY**

```
#define DEF_PRIORITY (20*HZ/100)
```

HZ je frekvence tiknutí, 100 Hz a tedy hodnota základního časového kvanta je 20 tiknutí
- deskriptor procesu obsahuje pro plánování položky:
- policy
  - plánovací třída, která povolené hodnoty
  - SCHED_FIFO - procesy reálného času, proces s nejvyšší prioritou má přidělen procesor jak dlouho chce
- SCHED_RR - cyklické plánovaní procesů reálného času
- SCHED_OTHER - konvenční procesy se sdílením času

- **rt_priority -** statická priorita procesů reálného času
- **priority -** základní časové kvantum
- **counter -** počet tiknutí, které procesu zůstávají do vypršení časového kvanta, na začátku je rovno trvání časového kvanta procesu, snižuje se po každém tiknutí v obsluze přerušení od časovače, když je proces vytvořen, hodnota **counter** se nastaví

```
rodič->counter >>= 1;
potomek = rodič->counter;
```

počet tiknutí se rozdělí na polovinu
- **need_resched -** příznak pro volání funkce **schedule()**

- vhodnost procesu pro jeho naplánování jako dalšího běžícího počítá funkce **goodness()** s parametry **běžící**, ukazatel na předcházející běžící proces a **p**, ukazatel na vyhodnocovaný proces

```
if (p = &init_task)
       return -1000;
if (p->policy != SCHED_OTHER)
       return 1000 + p->rt_priority;
if (p->counter == 0)
       return 0;
if (p->mm == běžící->mm)
       return p->counter + p->priority + 1;
return p->counter + p->priority;
```

- **p** získá malý bonus, sdílí-li adresový prostor s předcházejícím běžícím procesem
- plánovač realizuje funkce **schedule(),** její volání funkcemi v jádře může být přímé (*direct*) nebo odložené (*lazy*); přímé volání nastane, když běžící proces se musí stát spícím; odložené volání nastane, když běžící proces při návratu do uživatelského módu zjistí, že má příznak **need_resched** nastaven
- příznak **need_resched** je nastaven:
- když proces vyčerpal své časové kvantum

```
if (běžící->pid) {
    běžící->counter -= tiknutí;
    if (běžící->counter < 0) {
        běžící->counter = 0;
        běžící->need_resched = 1;
      }
```

```
}
```

- proces 0 nesmí být plánován pro sdílení času
- táto část obsluhy je odložená (*bottom half*), proto při snižování hodnoty **counter** se použije lokální proměnná **tiknutí**
  - když vhodnost vzbuzeného procesu je vyšší než běžícího

```
if (goodness(běžící, p) >
            goodness(běžící, běžící))
    běžící-> need_resched = 1;
```

  - když je zavoláno systémové volání **sched_setscheduler()** nebo **sched_yield()**, které umožňují nastavit strategii a prioritu nebo vzdát se procesoru procesům reálného času
- **schedule()** vybere nejvhodnější proces, kterému bude přidělen procesor
- proměnná **next** bude obsahovat ukazatel na deskriptor nejvhodnějšího procesu (s nejvyšší prioritou)
- napřed inicializujeme **next** na proces, od kterého začneme vyhodnocování a **c** na jeho vhodnost

```
if (běžící->stav == BĚŽÍCÍ) {
      next = běžící;
      if (běžící->policy & SCHED_YIELD) {
            běžící->policy &= ~SCHED_YIELD;
            c = 0;
      } else
            c = goodness(běžící, běžící);
} else {
    c = -1000;
    next = &init_task;
}
```

- jestli se běžící vzdal procesoru byl jeho příznak **SCHED_YIELD** nastaven a jeho vhodnost je 0
- začneme procesem, který byl dosud běžící nebo procesem 0 (**init_task**)
- projdeme kruhový seznam běhu schopných procesů a najdeme nejvhodnější proces

```
p = init_task.next_run;
while (p != &init_task) {
    váha = goodness(běžící, p);
    if (váha > c) {
        c = váha;
        next = p;
    }
    p = p->next_run
}
```

- je vybrán první proces s maximální váhou, předcházející běžící je upřednostněn před jinými běhu schopnými procesy se stejnou váhou
- je-li **c = 0** všechny běhu schopné procesy vyčerpali svoje časové kvantum a začíná nová epocha a **schedule()**přidělí všem procesům nové časové kvantum

```
if (!c) {
   pro_každý_proces(p)
      p->counter = p->priority + (p->counter >> 1);
```

- spícím a zastaveným procesům se tedy zvyšuje dynamická priorita

- nepreemtivní jádro Linuxu neposkytuje krátkou plánovací latenci a jeho využití pro reálny čas je omezené
- možná řešení:
  - zavedení bodů preempce (SVR4)
  - vytvoření preemtivního jádra (Solaris)
- efektivní plánovač pro reálny čas musí kromě plánovací latence řešit i další problémy

9.3.1 skryté plánování (*hidden scheduling*)

- proces reálného času mající vysokou prioritu, může vyžadovat službu jádra, kterou vykonává např. jádrové vlákno nižší priority

9.3.2 inverze priority (*priority inversion*)

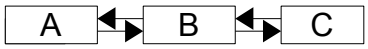
- proces vysoké priority musí čekat na uvolnění prostředku, který vlastní proces nízké priority

10 Synchronizace v jádře, symetrický multiprocesing

10.1 Problém synchronizace

Příklad:

1) přístup ke sdílené proměnné;
   - správnost výsledku závisi na plánování procesů - vzniká soutěž (*race condition*)
   - dále musí být zaručena nedělitelnost (atomicita) operace - tedy běh procesu nesmí být přerušen
2) odstranění deskriptoru procesuc;
   - deskriptory procesů jsou (cyklicky) obousměrně spojeny (obr. 1.1)
   - je-li proces odstaňování některého z procesů přerušen - údajová struktura -> nekonzistentní



obr. 1.1

- úsek kódu, který může být vykonáván současně nejvíce jedním procesem se nazává **kritická** o**blast/sekce** (*critical region/section*), jinak řečeno: proces, který začal vykonávat kritickou oblast, musí ji dokončit předtím než ji začne vykonávat jiný proces

10.2 Jádro

- kód jádra je vykonáván:
  - po volání systému
  - při obsluze výjimek
  - při obsluze přerušení

- jelikož jádro sdílí datové struktury způsobem uvedeným v příkladech, vzniká problém synchronizace výpočtů v jádře
- při prokládaném vykonávaní systémových volání a obsluh výjimek a přerušení musíme zabránit vzniku soutěže nad sdílenými daty

10.3 metody synchronizace

- jádro používá tyto metody:
  - nepreemtivnost procesů v módu jádro
  - atomické operace
  - zákaz přerušení
  - zamykání

10.3.1 nepreemtivnost procesů v módu jádro

- pokud je proces běžící v módu jádro, nemůže se vykonat preemce, tj. nemůže být nahrazen procesem s vyšší prioritou, pokud samotný proces neuvolní procesor
- proces v módu jádro může být přerušen obsluhou výjimky nebo přerušení; obsluha přerušení nebo výjimky může býž přerušena obsluhou přerušení

- neblokující systémová volání jsou atomická vzhledem k ostatním systémovým voláním, mohou bezpečně přistupovat k datům jádra, ke kterým nepřistupují obslužné programy přerušení a výjimek

10.3.2 atomické operace

- operace nad daty vykonaná jedinou instrukcí jsou na HW úrovni atomické
  - instrukce, které přistupují k paměti nejvíce jednou jsou atomické
  - čti/modifikuj/zapiš instrukce, které čtou data z paměti, modifikujíje a aktualizovaná data zapíšou zpátky do paměti jsou atomické, jestliže sběrnici mezi čtením a zápisem nezískal jiný procesor
  - čti/modifikuj/zapiš instrukce, kterých instrukční kód má prefix lock jsou atomické i na multiprocesorových systémech, řídící jednotka zamkne sběrnici dokud instrucke není dokončena

10.3.3 zákaz přerušení - maskování přerušení

- ne všechny operace nad daty můžeme vykonat atomickými operacemi. Data jádra, nad kterými pracují obsluhy přerušení můžeme efektivně zabezpečit tím, že při práci s nimi zakážeme přerušení
- (makra pro zakázaní přerušení - **cli** a pro povolení - **sti** ... makra nastavují a nulují **IF** příznak registru **eflags**)
- jádro při vstupu do kritické oblasti tedy nuluje příznak IF, na jejím konci ho však nemůže obecně přímo nastavit, protože vzhledem na možnost vnoření obsluh přerušení jádro neví jaký byl příznak IF před okamžitou obsluhou přerušení
- kritické oblasti, vytvořené zákazem přerušení musí být krátké, protože když do nich jádro vstoupí je blokována jakákoliv komunikace mezi V/V zařízeními a procesorem
- metoda zákazu přerušení není použitelná, když proces se stane spícím, protože je možné, že čeká na událost, která bude oznámena přerušením
- => řešením v těchto situacích je zamykání (jádrové semafory, kruhové blokování (*spin lock*))

10.3.4 zamykání

10.3.4.1 jádrové semafory (Linux)

- když se jádro pokusí o přístup k prostředku, který je obsazen jiným procesem, odpovídající proces přejde do stavu spící a stane se připravený, když je prostředek uvolněn
- jádrové semafory jsou objekty typu **struct semaphore**, který má položky:
  - **count** - uchovává celočíselnou hodnotu, je-li kladná prostředek je volný jinak je obsazen a absolutní hodnota udává počet čekajících požadavků jádra
  - **wait** - uchovává adresu seznamu čekajících
  - **waking** - zabezpečuje, aby jenom jeden proces po uvolnění prostředku mohl tento získat
- položka count je dekrementována, když se proces pokouší získat prostředek a inkrementována, když proces uvolní prostředek
- inicializace: MUTEX nastaví count na hodnotu 1 a MUTEX_LOCKED nastaví na hodnotu 0
- operace: P() a V()
- funkce: down() a up()
- požadavky se vykonávají v pořadí adres datových struktur **semaphore**

- **multiprocesing**
  - v jednoprocesorových strojích má paralelizmus procesů tvar prokládání jejich činností
  - na víceprocesorových strojích můžou být činností procesů až do počtu procesorů vykonávány současně
- **SMP** (symmetrical multiprocessing) architektura
  - procesory a sdílená hlavní pamět jsou připojeny ke společné sběrnici. Synchronizace mezipaměti (cache) procesorů, když procesor modifikuje svou mezipamět, musí kontrolovat jestli stejná data nejsou v mezipaměti jiného procesoru a když ano, musí je aktualizovat nebo zneplatnit.
- instrukce čti/modifikuj/zapiš na multiprocesorových systémech nejsou atomické, sběrnice se musí zamkout - **lock**
- SMP jádro
  - tradiční řešení synchronizace
    - neblokující systémová volání jsou atomická vzhledem k ostatním systémovým voláním
    - pro synchronizaci prokládáním obsluh přerušení při přístupu ke společným datům stačí maskovat přerušení
  - multiprocesorový systém
    - na více procesorech můžou být současně vykonávána systémová volání
    - na více procesorech můžou být současně obsluhována přerušení i když každý z nich maskuje přerušení
  - řešeni: semafory a kruhové blokování na chránění přístupu ke sdíleným datům, tj. na implementaci kritických sekcí

10.3.4.2 kruhové blokování

- instrukce test-and-set zjistí hodnotu sdílené proměnné, 0 nebo 1, a nastaví ji na hodnotu 1
- lock = = 0 ... prostředek je volný, možno vstoupit do kritické oblasti, proces může pokračovat
- lock = = 1 ... prostředek je obsazen, do kritické oblasti není možno vstoupit, proces musí čekat
- kruhové blokování je neefektivní na jednoprocesorových systémech, protože nemá kdo splnit podmínku, na kterou se čeká
- je efektivní na multiprocesorových systémech, protože řešení se semaforem, vyžaduje přepnutí kontextu, co je nákladné
- **spin_lock** a **spin_unlock** můžou chránit jenom data jádra, ke kterým nikdy nepřistupují obslužné programy přerušení
- kruhové blokování pro čtení/zápis (*read/write spin locks*)
  - přístup ke sdíleným datovým strukturám v jádře nemusí být nevyhnutelně exklusivní
  - čtení je možné povolit současně několika výpočtům v jádře
  - je-li sdílená datová struktura zamčena pro čtení, můžou i další výpočty v jádře tuto strukturu číst, nemůžou však do ni zapisovat
  - možnost současného čtení zvyšuje pralelnost výpočtů jádra
  - oba zámky jsou realizovány kruhovým blokováním pro čtení/zápis
  - jsou implementovány 32 bitovým počítadlem, reprezentující okamžitý počet výpočtů jádra, které čtou chráněnou datovou strukturu
  - jeho bit v nejvyšším řádu slouží pro kruhové blokování zápisu, je nastaven modifikuje-li jádro datovou strukturu

11 Meziprocesová komunikace, roury, semafory, fronty zpráv, sdílená paměť

- mnohé aplikace sestávají z mnoha navzájem spolupracujících procesů, které mezi sebou komunikují a sdílejí informace
- jádro musí poskytovat mechanizmy, které toto umožní
- nazýváme je prostředky meziprocesové komunikace
- jejich účelem je:
  - přenos údajů
  - sdílení dat
  - oznámení vzniku událostí
  - sdílení prostředků
  - sledování a řízení běhu procesu, např. při ladění programů

- **signály** - umožňují oznámit procesům asynchronní události
- **roury (pipes)** - umožňuje zapisovat data na konec roury a číst je ze začátku
- **nepojmenované roury** - vytvářejí systémovým voláním pipe(), které vrátí dva deskriptory jeden pro čtení a druhý pro zápis. Deskriptory roury jsou při vytváření procesů děděné, do roury může zapisovat a číst z ní více procesů, přičemž data jsou čtena v pořadí v jakém byla zapsána

- procesy můžou komunikovat prostřednictvím roury byla-li vytvořena společným předchůdcem, po skončení všech procesů roura přestává existovat

11.1 pojmenované roury, FIFO soubory

- jsou perzistentní, existují jako soubory i když je nepoužívají žádné procesy
- FIFO musí být explicitně zrušen, jako obyčejné soubory - unlink
- na rozdíl od obyčejných souborů přečtená data jsou odstraněna a z pohledu komunikace mají stejnou sémantiku jako nepojmenované roury

vytvoření FIFO souboru: *mknode (cesta, mod, zařízení)*
mkfifo (cesta, mod)
mod - obsahuje obvyklá oprávění

11.2 sledování procesů

- systémové volání: *ptrace (příkaz, pid, adr, data)*
  - umožňuje sledovat a řídit běh procesu **pid**
  - příkaz = = 0 , slouží na informování jádra, že proces je sledován
  - ostatní příkazy používá sledující proces na řízení vykonávání sledovaného procesu

11.3 Systém V IPC

- předcházející mechanizmy meziprocesorové komunikace nejsou pro mnohé aplikace postačující
- systém V poskytl tři mechanizmy : (postupně byly implementovány v dalších systémech - BSD, Linux)
  - semafory (*semaphores*)
  - fronty správ (*message queues*)
  - sdílenou paměť (*shared memory*)

- společné data IPC prostředku (semafor, fronta správ, sdílená pamět) jsou uložena v záznamu **ipc_perm**, který obsahuje položky:
  - **key** - klíč, 32 bitová hodnota zadaná uživatelem, který identifikuje konkrétní prostředek
  - **uid** - uživatelský ID vlastníka prostředku
  - **gid** - skupinový ID vlastníka prostředku
  - **cuid** - uživatelský ID tvůrce prostředku
  - **cgid** - skupinový ID tvůrce prostředku
  - **seq** - pořadové číslo použití záznamu prostředku
- IPC datové struktury se vytvářejí dynamicky a jsou perzistentní
- proces získá IPC prostředek voláním **semget()**, **msgget()** a nebo s**hmget()**
- prvním parametrem je klíč, který procesy identifikují prostředek a uvedené funkce vrátí identifikátor prostředku, který procesy dál používají pro přístup k prostředku
- identifikátor prostředku, vzhledem ke své perzistentnosti, není přímo index do tabulky prostředků, vypočte se podle vztahu: *id = seq * velikost_tabulky + index*
- naopak, jádro z identifikátoru prostředku, který je parametrem dalších systémových volání, určí index prostředku v tabulce podle vztahu: *index = id % velikost_tabulky*
- použité parametry ve funkcích xxxget:
  - IPC_PRIVATE ... zajistí vytvoření nového prostřeku
  - IPC_CREATE ... způsobí vytvoření prostředku jádrem (pokud ještě neexistuje)
  - IPC_EXCL ... jádro oznámí chybu jestliže prostředek se zadaným klíčem existuje
  - IPC_SET a IPC_STAT ... umožní nastavit a zjistit stavové informace prostředků
  - IPC_RMID ... v xxxctl odstraní IPC prostředek
  - IPC_NOWAIT ... jádro namísto blokování vrátí chybu

11.4 semafory

semid = semget (klíč, počet, příznak) ... vrátí pole semaforů o velikosti počet

stav = (semid, sops, nsops) ... *sops* je ukazatel na pole s *nsops* prvky typu *sembuf* specifikující operaci *sem_op* nad semaforem s indexem *sem_num*

```
struct sembuf {
        unsigned short sem_num;
        short sem_op;
        shor sem_flg;
}
```

sem_op < 0 ... je-li hodnota semaforu větší nebo rovná abs. hodnotě sem_op, abs. hodnota sem_op se odečte od hodnoty semaforu; je-li menší, proces je blokován (spící) dokud není zvýšena hodnota semaforu

sem_op > 0 ... zvíší se hodnota semaforu o hodnotu sem_op a vzbudí se procesy čekající na její zvýšení

s**em_op = 0** ... proces je blokován dokud hodnota semaforu není 0

- jde tedy o zobecněný semafor
- je-li proces spící uprostřed operace, je spící na přerušitelné úrovni

11.5 fronty zpráv

- procesy komunikují prostřednictvím zpráv
- zpráva vytvořená procesem je zaslána do fronty zpráv dokud ji jiný proces nepřečte
- zpráva obsahuje 32 bitový typ zprávy a data zprávy
- typ zprávy umožňuje selektivně vybírat zprávy z fronty
- proces získá nebo vytvoří frontu zpráv voláním
  msgqid = msgget (klíč, příznak)

- zpráva se uloží do fronty voláním: *msgsnd (msgqid, msqp, počet, příznak)*
  msgp - je ukazatel na zprávu obsahující typ zprávy následován daty
  počet - velikost zprávy včetně typu v bytech
- zprávy jsou ve frontě v pořadí jejich příchodu
- zprávy jsou vybírany voláním : *počet = msgrcv (msgqid, msgp, maxpct, msgtyp, příznak)*
- je-li čtena zpráva delší než *maxpct* zpráva je useknuta
- je-li *msgtyp* nulový, vrátí se první zpráva z fronty
- je-li *msgtyp* záporný, vrátí se první zpráva nejnižšího typu než je abs. hodnota *msgtyp*

11.6 sdílená paměť

- sdílená paměť je oblast paměti, která je sdílená více procesy
- proces sdílenou oblast pamětí vytvoří nebo ji získá voláním :
  shmid = (klíč, velikost, příznak)
- procesy potom připojí oblast na virtuální adresu voláním
  adr = shmat (shmid, shmadr, shmpříznak)
  shmadr - je návrh adresy pro připojení oblasti
  shmpříznak SHM_RND - způsobí zaokrouhlení adresy dolů
  shmadr - je-li nula, jádro vybere adresu - skutečná adresa je návratová hodnota
- odpojení oblasti:
  shmdt (shmaddr)

12 Virtuální souborový systém

- operační systém musí poskytovat prostředek pro perzistentní uložení dat a jejich správu
- soubor kontejneru pro data
- souborový systém umožňuje organizaci souborů a přístup k nim
- tradiční souborový systém Systém V file systém, s5fs, původně i v BSD systémech, byl součástí jádra

12.1 BSD Fast File Systém - FFS

- souborový systém byl dále součástí monolitického jádra a ani s5fs a FFS nemohly koexistovat v jednom operačním systému
- navíc cenná data můžou být uložena i v souborových systémech jiných operačních systému (Win)
- operace nad soubory, systémová volání, se "jenom" různě vykonávají
- VFS je vrstva jádra obsluhující všechna systémová volání pro souborový systém a poskytuje rozhraní současně pro různé souborové systémy
- souborové systémy podporované VFS můžeme rozdělit do třech skupin:
  a) diskové souborové systémy
     - s5fs, FFS, Ext2(Linux)
     - MS-DOS, Windows
     - ISO9660 CD-ROM souborový systém
     - ostatní
  b) síťové souborové systémy
     - Network File Systém, NFS
     - SMB (Microsoft)
     - NCP, NetWare Core Protocol (Novell)
  c) speciální souborové systémy
     - nespravují diskový prostor, devfs (Linux)

12.2 Linux VFS

- zavádí obecný souborový model schopný reprezentovat všechny podprorované souborové systémy
- zrcadlí tradiční souborový systém operačního systému Unix s cílem minimální režie pro nativní souborový systém
- jádro nemůže přímo obsahovat kód pro jednotlivé funkce jako je **read()**, namísto toho pro každou operaci použije ukazatel na imlementující funkci pro daný souborový systém
- souborový systém je po otevření ve VFS reprezentován datovou strukturou **file**, která obsahuje položku **f_op**, která obsahuje ukazatel na funkce specifické pro konrétní typ souborového systému
- pro operaci **read()** z příkladubude položka **f_op** údajové struktury **file** obsahovat ukazatel na funkce pro MS-DOS a volání funkce **read** je nepřímé
  file->f_op->read(...);
- pro operaci **write()** položka **f_op** údajové struktury **file** bude obsahovat ukazatel na funkce pro Ext2
- na model můžeme nahlížet objektově
- objekty jsou implementovány jako záznamy s položkami obsahující data a položkami obsahující ukazatele na funkce, odpovídající metodám objektu

- obecný souborový model se skládá z objektů následujících typů:
  - **objekt superblok -** uchovává globální informace o souborovém systému
  - **objekt iuzel (vuzel) -** uchovává informace o jednotlivém souboru, číslo iuzlu jednoznačně identifikuje soubor v souborovém systému
  - **objekt soubor -** uchovává informace o interakci mezi otevřeným souborem a procesem
  - **objekt položka adresáře** (*directory entry, dentry*) - uchováva odkaz na soubor odpovídající položce adresáře, uložení této informace na disku se pro jednotlivé typy souborových systémů liší
- VFS obsahuje mezipaměť nedávno použitých položek adresáře (*dentry cache*), urychluje převod cesty v adresáři na iuzel poslední součásti cesty
- některá systémová volání nevyžadují volání specifických funkcí konkrétního systémového souboru
- například **`lseek()`**, které nastavuje pozici v souboru pro další operaci, co je atribut, který se vztahuje k interakci otevřeného souboru a procesu, vyžaduje modifikaci jenom odpovídajícího objektu typu soubor a je tedy nezávislé na typu souborového systému

12.2.1 objekt superblok

- všechny objekty superblok (jeden pro začleněný systém) jsou spojeny v obousměrném spojovém seznamu
- údaje v položce **`u`**, například bitová mapa přidělených bloků, jsou kopírovány do paměti, jsou-li tyto údaje změněny superblok na disku se musí aktualizovat
- metody objektu superblok jsou v záznamu **`super_operations`**, kterého adresa je v poli **`s_op`**
- VFS potřebné operace, například přečtení iuzlu **`read_inode()`**, volá **`sb->s_op->read_inode();`**
- příklady operací superbloku:
  - **`read_inode(inode)`** - čte údaje z iuzlu na disku a vyplní položky objektu iuzel
  - **`write_inode(inode)`** - aktualizuje iuzel na disku z obsahu položek objektu iuzel
  - **`delete_inode(inode)`** - odstraní datové bloky obsahující soubor, diskový iuzel a VFS iuzel
  - **`put_super(super)`** - uvolní objekt superblok (odpovídající souborový systém je odčleněn)
- v operacích se nenachází metoda **`read_super`** pro čtení superbloku z disku
- uvedené metody jsou metody objektu superblok pro začleněný souborový systém
- metoda read_super je metodou reprezentující typ souborového systému, která je volaná při začleňovámí souborového systému systém

12.2.2 objekty iuzel

- všechny informace o souboru potřebné pro práci souborového systému se souborem jsou v iuzlu, speciálně jde o ukládání a vybírání informací uložených v souboru
- jméno souboru je více nebo méně náhodné označení souboru
- jedinečná reprezentace souboru je iuzel

- každý inode objekt je vždycky v jednom ze tří obousměrných spojových seznamů

- seznam nepoužívaných (volných) iuzlů
- seznam používaných iuzlů
- seznam modifikovaných iuzlů

- vytvořených položkou **`i_list`**
- objekty inode v seznamech používaných a modifikovaných iuzlů jsou také v rozptýlené tabulce
  - **`create(dir, dentry, mode)`** - vytvoří nový diskový iuzel pro obyčejný soubor sdružený s objektem položka adresáře
  - **`lookup(dir, dentry)`** - hledá v adresáři iuzel odpovídající jménu souboru v objektu položka adresáře
  - **`link(old_dentry, dir, new_dentry)`** - vytvoří nový odkaz na soubor specifikovaný parametrem **`old_dentry`** v adresáři **`dir`** , jméno nového odkazu je v **`new_dentry`**
  - **`unlink(dir, dentry)`**
  - **`symlink(dir, dentry, symname)`**
  - **`mkdir(dir, dentry, mode)`**
  - **`rmdir(dir, dentry)`**
  - **`mknod(dir, dentry, mode, rdev)`**

12.2.3 objekty soubor

- objekt soubor opisuje interakci procesu s otevřeným souborem
- je vytvořen když je soubor otevřen a vytváří ho záznam **`file`** s položkami

`f_next`	ukazatel na další objekt soubor
`f_pprev`	ukazatel na předházející objekt soubor
`f_dentry`	ukazatel na združený objekt položka adresáře
`f_op`	ukazatel na tabulku operací
`f_mode`	mód přístupu k souboru
`f_pos`	pozice v souboru
`f_count`	počet použití objektu

- objekty soubor nemají odpovídající obraz na disku a proto nemají položku, do které se zaznamená jejich modifikace
- každý objekt soubor je na jednom ze dvou obosměrných kruhových seznamů
  - seznam nepoužívaných objektů soubor, položka **`f_count`** je nulová
  - seznam používaných objektů soubor

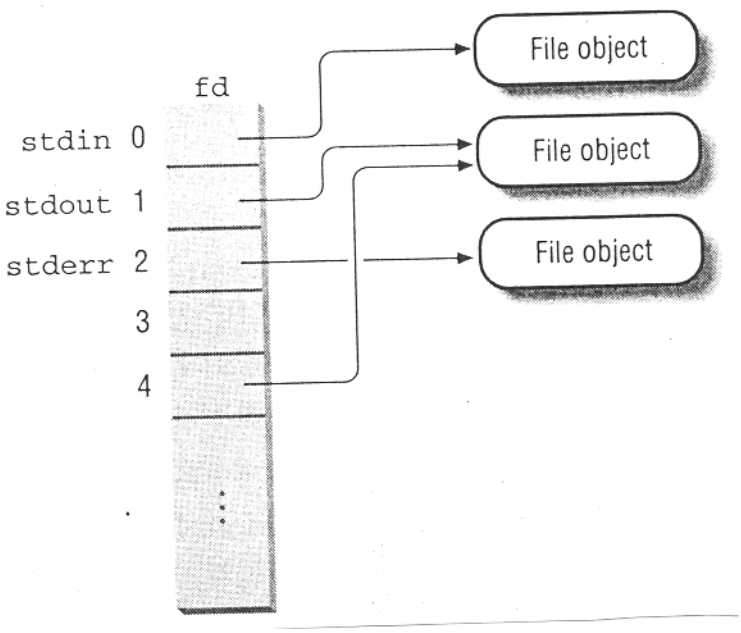
12.2.4 objekty položka adresáře

- v modelu je adresář soubor obsahující seznam souborů a adresářů
- po přečtení položky adresáře je transfomována na objekt položka adresáře
- objekt položka adresáře se vytvoří pro každou část cesty v adresáři
- pro /tmp/test se vytvoří tři objekty položka adresáře
- objekt nemá odpovídající strukturu na disku, neobsahuje tedy položku pro zaznamenání modifikace
- objekty položka adresáře se uchovávají v mezipaměti objektů položka adresáře (*dentry cache*)

- pro zrychlení přístupu k objektům položka adresáře se používá rozptýlená tabulka (*hash table*), přičemž hodnota rozptylové funkce se vytváří z adresy objektu položka adresáře adresáře a jména souboru

12.2.5 soubory sdružené s procesem

- každý proces má svůj pracovní adresář a kořenový adresář
- adresa záznamu s touto informací je v deskriptoru procesu
- vzhledem např. na systémové volání dup() můžou dva deskriptory souboru odkazovat na stejný objekt soubor



12.2.5.1 připojení souborového systému

- před začátkem používání souborového systému se musí vykonat dvě operace
  - registrace
  - připojení (začlenění)
- registrace se vykoná buď při zavádění operačního systému (*boot*) nebo při zavedení modulu implementujícího souborový systém
- po registraci souborového systému má jádro k dispozici jeho specifické funkce a souborový systém takového typu může být připojen
- souborový systém, kterého kořenový adresář je kořenem systémového stromu adresářů se nazývá kořenový souborový systém
  - ostatní souborové systémy můžou být připojeny k systémovému stromu adresářů
  - adresáře, na které jsou připojeny souborové systémy se nazývají body připojení

12.2.5.2 registrace souborového systému

- Linux je možné konfigurovat tak, aby rozeznával všechny potřebné typy souborových systémů při překladu jádra
- kód implementující souborový systém může být zaveden také dynamicky jako modul

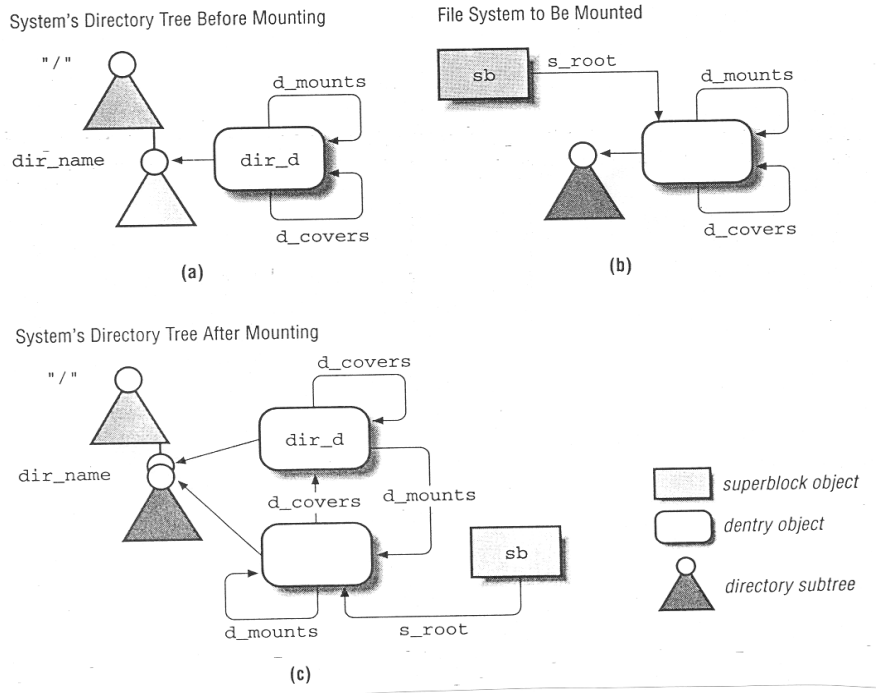
12.2.5.3 připojení kořenového souborového systému

- z (diskového) zařízení se pokouší přečíst superblok voláním metody **`read_super`** registrovaných objektů **`file_system_type`**
  - pro / vytvoří objekt iuzel a objekt položka adresáře

- nastaví položky **`root`** a **`pwd`** procesu **`init`** na objekt položka adresáře /
- vloží první prvek do seznamu připojených souborových systémů

12.2.5.4 připojení všeobecného (generického) souborového systému

- standardní tvar příkazu pro připojení
  `mount -t typ zařízení adresář`
  - odevzdá jádru fyzické zařízení, na kterém je souborový systém a jeho typ a adresář kam bude ve stromu adresářů připojen nový souborový systém
  - předcházející obsah tohoto adresáře (pokud nějaký byl) se stane neviditelný dokud nový souborový systém zůstane připoj



12.2.5.5 odpojení souborového systému

- postup je v zásadě opačný
- nelze odpojit souborový systém, kterého soubory jsou používány
- nelze odpojit kořenový systém souborů
- změněné objekty se zapíšou na disk

12.2.5.6 prohledávání cesty k souboru

- cílem je, aby VFS ze zadané cesty k souboru určil odpovídající iuzel
- cesta se rozdělí na posloupnost jmen souborů, které všechny, kromě posledního, musí být adresáře
- je-li začáteční znak /, cesta je absolutní a prohledávání začne adresářem
  `běžící->fs->root`
- jinak prohledávání začne v adresáři
  `běžící->fs->pwd`
- následně se hledá v adresáři položka s prvním jménem v cestě, čím se získá iuzel prvního adresáře v cestě
- postup se opakuje až projdeme celou cestu

- celý naznačený postup značně urychluje mezipaměť objektů položka adresáře
- přitom nutno vzít v úvahu
  - přístupová páva pro každý adresář
  - jméno může být symbolický odkaz a postup musí pokračovat pro všechny části cesty v symbolickém odkazu
  - vznikne-li symbolickými odkazy kruh, musí být identifikován a prohledávání skončit chybou
  - jméno může být bod připojení souborového systému a prohledávání musí pokračovat v novém souborovém systému

12.2.5.7zamykání souborů (file locking)

- operační sytém UNIX byl navržen se souběžným přístupem k souborům více procesy
- obdobně jako u sdílených proměnných vzniká problém synchronizace
- POSIX požaduje mechanizmus zamykání umožňující zamknout libovolnou část souboru – od jednoho bytu až celý soubor
- jelikož je možno zamykat soubor po částech proces může vlastnit několik zámků
- pokud je nějaká část souboru zamknuta a jiný proces nekontroluje její zamčení může jiný proces k zamčené části přistoupit
- takovéto zámky se nazývají poradní (*advisory locks*) a vyžadují spolupráci procesů
- jsou implementovány na bázi systémového volání **`fcntl()`**

- tradiční varianty BSD implementují poradní zámky systémovým voláním **`flock()`**, které však neumožňuje zamykání částí souboru, ale jenom celý soubor
- tradiční varianty Systému V poskytují systémové volání **`lockf()`**, co je jenom rozhraní k **`fcntl()`**
- navíc System V R3 zavedl mandatorní (*mandatory*) zámky, kdy jádro kontroluje zamčení souboru při každém volání **`open(), read(), write()`**

- bez ohledu na to, jestli procesy používají poradní nebo mandatorní zámky mohou využívat
  - sdílené (*shared*) zámky pro čtení
  - výhradní (*exclusive*) zámky pro psaní

- libovolný počet procesů může vlastnit sdílené zámky, ale jenom jeden proces může vlastnit výhradní zámky

okamžitý zámek	požadovaný zámek	
	čtení	zápis
žádný	ano	ano
čtení	ano	ne
psaní	ne	ne

13 Implementace souborového systému

diskový blok, sektor
disková oblast (*disk partition*) posloupnost po sobě následujících očíslovaných diskových bloků stejné velikosti

System V File System (1978)

- první univerzální implementace souborového sytému
- jednoduchý návrh
- boot blok, super blok, tabulka iuzlů, datové bloky
- superblok jeden
- jedna tabulka diskových iuzlů a jedna oblast datových bloků
- přidělování diskových iuzlů náhodné, tj. iuzly souborů téhož adresáře nejsou seskupeny
- přidělování diskových bloků suboptimální, jenom při vytvoření souborového systému v diskové oblasti je seznam volných bloků konfigurován rotačně po sobě
- seznam (omezený) volných iuzlů v superbloku, po jejich vyčerpání, čtení bloků s iuzly
- seznam volných datových bloků jako seznam volných bloků s čísly volných bloků

BSD Fast File System (1984)

- stejná funkcionalita
- hlavní přínos v rozvržení disku
- kromě rozdělení disku na oblasti, které obsahují souborové systémy, rozděluje oblast dále na skupiny malého počtu válců (*cylinder group*), které obsahují bloky se souvisejícími iuzly a datovými bloky
- fragmentace bloků, blok může být rozdělen na 1, 2, 4, 8 fragmentů s nejmenší velikostí rovnající se velikosti sektoru
- bloky souboru jsou uloženy v diskových blocích, kromě posledního bloku, který může obsahovat jeden nebo více po sobě následujících fragmentů

UNIX File System - UFS

- SunOS 2.0, Solaris vychází z BSD FFS

Second Extended Filesystem – Ext2

- první verze operačního sytému Linux vycházely ze souborového sytému operačního systému Minix
- později byl vytvořen Extended Filesystem – Ext FS a v roce 1994 byl uveden Ext2

efektivnost Ext2

- při vytváření souborového systému je možné specifikovat velikost bloku (1024 až 4096 bytů), jestliže očekáváme soubory s několika tisíci bytů volíme velikost 1024,

čím snižujeme interní fragmentaci (průměrně není využito půl bloku), na druhé straně veliké bloky pro rozsáhlé soubory snižují počet diskových operací

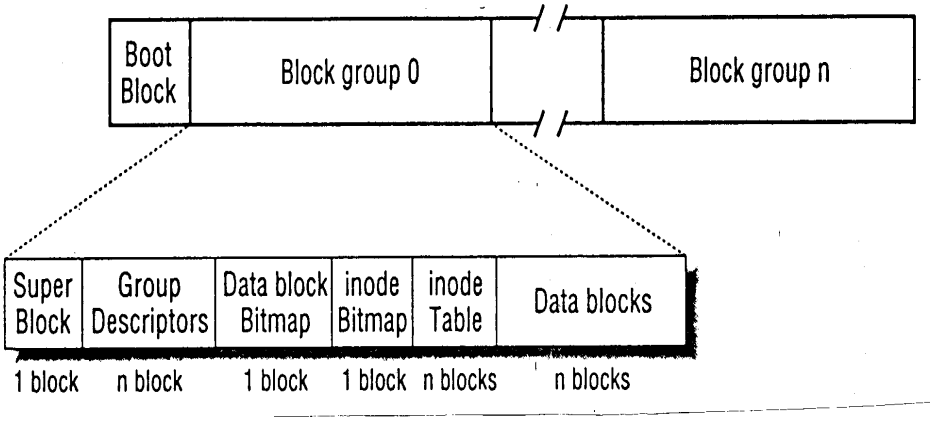
- pro diskovou oblast můžeme zadat povolený počet iuzlů podle očekávaného počtu souborů, co maximalizuje využití diskového prostoru
- souborový systém je rozdělen na skupiny bloků, každá skupina obsahuje bloky na sousedních stopách
- souborový systém předem přiřadí (*preallocates*) bloky obyčejným souborům, tj. předtím než jsou skutečně požadovány, při zvětšení souboru jsou tak dispozici sousedící diskové bloky
- podporuje rychlé symbolické odkazy, má-li symbolický odkaz 60 znaků nebo méně, je uložen v iuzlu

robustnost a flexibilita

- aktualizace souborů je navržena s ohledem na minimalizaci škody v případě havárie diskového systému, například při vytváření nového odkazu na soubor, napřed se zvýší počet odkazů v iuzlu a nové jméno se uloží do příslušného adresáře až následně
- podporuje automatickou kontrolu konzistenci souborového sytému, při zavádění systému, po předdefinovaném počtu připojení souborových systémů nebo po uplynutí předdefinovaného času od poslední kontroly
- podpora neměnných souborů
- podpora SVR4 i BSD sémantiky pro GID nového souboru, v SVR4 má nový soubor GID procesu, který ho vytvořil, v BSD nový soubor získá GID podle adresáře, ve kterém je vytvořen

diskové datové struktury

- první blok každé diskové oblasti obsahuje zaváděcí program (*boot sector*)
- zbytek je rozdělen na skupiny bloků s rozvržením podle obrázku



blok skupiny bloků obsahuje

- kopii superbloku
- kopii skupiny deskriptorů skupiny bloků
- bitovou mapu datových bloků
- bitovou mapu iuzlů
- skupinu iuzlů

- kousek dat patřících souboru, datový blok

  - o blocích neobsahujících data souboru říkáme, že obsahují metadata
  - pokud blok neobsahuje data souboru ani metadata nazývá se volný
  - jádro používá superblok a deskriptory skupin bloků jenom ze skupiny bloků 0, ostatní jsou udržovány jako kopie a můžou být použity v případě havárie souborového systému
  - počet skupin bloků v diskové oblasti je omezen požadavkem, aby se bitová mapa bloků skupiny vešla do jednoho bloku
  - velikost skupiny bloků může být nejvíc *8*b* bloků, kde *b* je velikost bloku v bytech
  - jeli *o* velikost diskové oblasti v blocích počet skupin bloků je přibližně *o/(8*b)*

příklad

o = 8 GB, b = 4KB
bitová mapa opisuje využití 32 KB bloků
oblast obsahuje 8 GB / 4 KB = 2 MB bloků
je třeba nejmíň 2 MB / 32 KB = 2^{11} / 2^{5} = 64 skupin bloků

superblok
obsahuje údaje pro správu celého souborového systému

deskriptor skupiny bloků
záznam velikosti 32 bytů, posledních 14 nevyužito, obdoba superbloku pro skupinu bloků

bitové mapy
posloupnost bitů, ve které hodnota 0 specifikuje, že odpovídající blok nebo iuzel je volný a hodnota 1 specifikuje, že je používán

každá bitová mapa musí být uchována v jednom bloku velikosti 1024, 2048 nebo 4096 bytů, jedna bitová mapa opisuje 8192, 16 384, nebo 32 768 bloků

tabulka (pole, seznam) iuzlů
každý iuzel má velikost 128 bytů, blok o velikosti 1024 tedy obsahuje 8 iuzlů, ...

použití diskových bloků různými typy souborů

- obyčejné soubory
- adresář
- symbolické odkazy
- soubory typu zařízení, pojmenovaná roura, socket

datové struktury v paměti

většina informací uložených v diskových datových strukturách je při jejich používání kopírována do mezipaměti v hlavní paměti

typ	uložení v mezipaměti
superblok	stále v mezipaměti
deskriptor skupiny	stále v mezipaměti

bitová mapa bloků	pevné omezení
bitová mapa iuzlů	pevné omezení
iuzel	dynamické
datový blok	dynamické

- data, která jsou často aktualizována jsou v mezipaměti stále
- ve vyrovnávacích pamětech bloků (*buffer cache*) se uchovává pevný počet bitových map bloků a iuzlů, nejstarší jsou přepsány na disk kdyby jejich počet měl překročit omezení
- iuzly a datové bloky jsou uchovávány ve vyrovnávacích pamětech bloků dokud je sdružený objekt používán, po skončení používání můžou být přepsány na disk

mezipaměti bitových map

- vyrovnávací paměť bloku obsahující superblok připojeného souborového systému Ext2 se uvolní jenom když se souborový systém odpojí
- všechny bitové mapy vzhledem na jejich počet není možné stále uchovávat v hlavní paměti

příklad

velikost disku 4 GB
velikost bloku 1 KB

bitová mapa v jednom bloku opisuje $8*2^{10}$ bloků = 8 MB
nejmenší počet skupin bloků je $4*2^{10}$ MB / 8 MB = 512
velikost bitových map bloků a iuzlů v jedné skupině bloků je 2 KB
na mezipaměť všech bitových map je třeba 1 MB paměti

správa diskového prostoru

přidělování diskových iuzlů

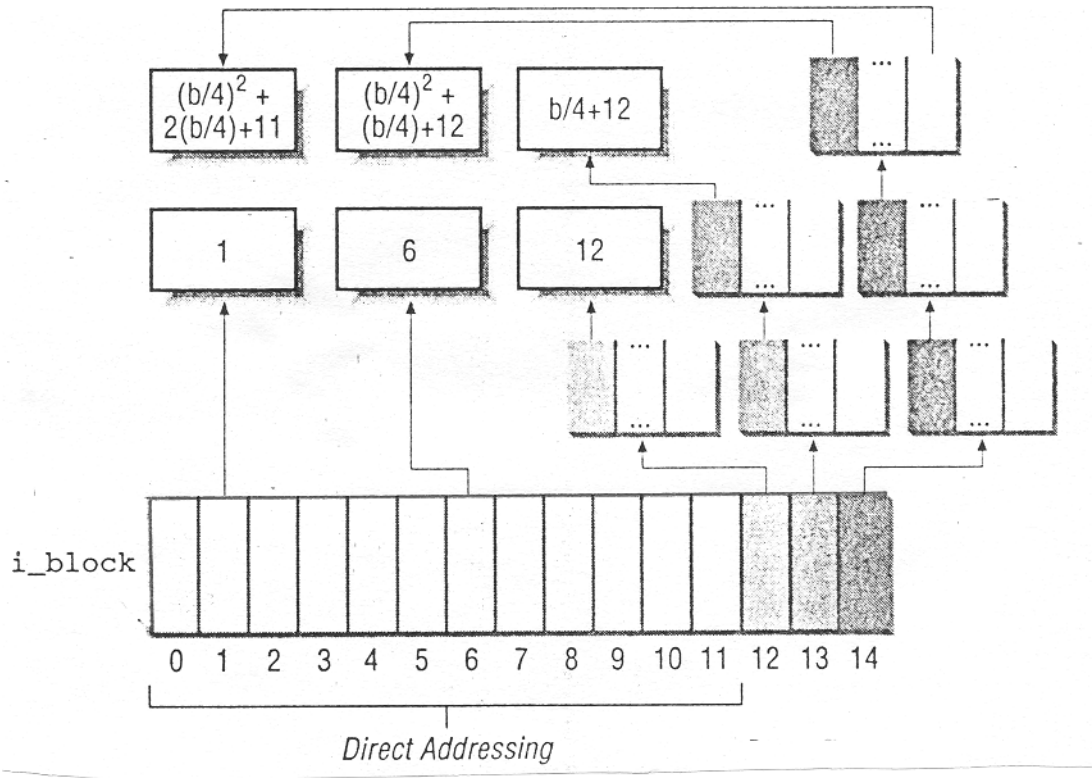
- je-li nový iuzel adresářem je snahou, aby jsme udrželi stejnoměrné rozložení adresářů v částečně zaplněných skupinách bloků
- přidělí se iuzel ze skupiny bloků, která má největší počet volných bloků ze všech skupin bloků s počtem volných iuzlů větším než je průměrný počet volných bloků
- nenalezne-li se volný iuzel prohledávají se skupiny bloků sekvenčně od první skupiny bloků

adresování datových bloků

- neprázdné obyčejné bloky sestávají z několika datových bloků
- tyto bloky můžeme označovat pořadím v souboru, čísla bloků souboru, nebo čísly diskových bloků souborového systému
- obecně přistupujeme k souboru od pozice f a nutno určit, ve kterém diskovém bloku se byte na této pozici v souboru nachází
- z pozice f se určí *číslo bloku souboru = f / velikost bloku*

číslo bloku souboru se převede na číslo diskového bloku, ve kterém se nachází, využitím pole **`i_block`** v diskovém iuzlu

- prvních 12 prvků obsahuje čísla diskových bloků, které obsahují prvních 12 bloků souboru, které mají čísla bloků souboru od 0 do 11
- prvek s indexem 12 obsahuje číslo diskového bloku, obsahující další pole čísel diskových bloků, je-li velikost bloku *b* a číslo diskového bloku je umístněno ve 4 bytech, potom jsou to čísla diskových bloků dalších bloků souboru, které mají čísla od 12 do *b/4 + 11*
- prvek s indexem 13 obsahuje číslo diskového bloku, který obsahuje čísla diskových s čísly diskových bloků pro následující bloky souboru
- *prvek s indexem 14 obsahuje číslo diskového bloku, ...*



mezery v souborech (*file holes*)

aplikace využívající rozptýlení v souborech vytvářejí v souboru mezery, pro bloky souboru, které jsou v mezeře není třeba přidělit diskové bloky, ty si přidělí až se bude do bloku skutečně zapisovat

přidělování datových bloků

- s cílem snížit fragmentaci souboru Ext2 zkouší přidělit nový blok nejblíže k posledně přidělenému bloku
- jeli neúspěšný, hledá nový blok v skupině bloků, ve které je iuzel souboru
- naposledy hledá v jiné skupině bloků
- souborový systém Ext2 přiděluje datové bloky předem
- pro přidělení nového bloku se nejdřív určí cíl

14 Správa V/V zařízení

sdružení souboru a V/V zařízení

můžeme psát do souboru nebo poslat znak na tiskárnu psaním na /dev/lp0

soubor typu zařízení

- druh
  blokové nebo znakové
- hlavní číslo (*major number*)
  číslo identifikující typ zařízení, zařízení se stejným hlavním číslem a stejného druhu sdílejí stejnou množinu operací a jsou obsluhovány stejným drajvrem
- vedlejší číslo
  identifikuje specifické zařízení ve skupině zařízení se stejným hlavním číslem

systémové volání **`mknod()`** s parametry jméno, druh, hlavní a vedlejší číslo vytváří soubory typu zařízení:

- **bloková zařízení** - přenášejí blok dat pevné velikosti jednou V/V operací k blokům zařízení je libovolný přístup
- **znaková zařízení** - přenášejí data různé velikosti jednou V/V operací k datům je přístup sekvenční
- **síťová zařízení** - nemají odpovídající soubor typu zařízení

aplikační programy nepoužívají obvyklá souborová systémová volání, ale volání zavedená BSD, **`socket(), bind(), listen(), accept(), connect()`**

práce VFS se soubory typu zařízení

namísto souborových systémových volání požadované funkce jsou vykonány samotným zařízením, ty jsou vykonávány drajvrem zařízení

v položkách tabulek **`chrdevs`** a **`blkdevs`** se nachází ukazatel na operace nad souborem typu zařízení, hlavní číslo zařízení je indexem do tabulky

obsluha znakových zařízení

data nejsou přenášena přes vyrovnávací paměť, ale přímo do uživatelského adresového prostoru, znakové rozhraní můžou mít i disky, například pro databázové aplikace (*raw V/V*)

zpracování řádky, síťové protokoly obsahují části nezávislé od zařízení, možné funkční rozdělení, podporováno mechanizmem proudu (*stream*), co je plně duplexní zpracování a přenos dat mezi jádrovým adresovým prostorem drajvru a uživatelským prostorem procesu

obsluha blokových zařízení

dva druhy přenosu dat:

- V/V operace s vyrovnávací pamětí

- stránkové V/V operace

V/V operace s vyrovnávací pamětí

- ve vyrovnávací paměti je uložen diskový blok
- vyrovnávací paměti bloků jsou organizovány v mezipaměti vyrovnávacích pamětí bloků (*block buffer cache*) pomocí hlaviček bloků

stránkové V/V operace

- přenos dat po blocích, adresový prostor procesu je množina stránek
- V/V operace pro obyčejné soubory jsou vykonávány a ukládány po stránkách v mezipaměti stránek