**PB153 OPERAČNÍ SYSTÉMY A JEJICH ROZHRANÍ - POZNÁMKY  
Poznámka autora: Ide o rozšírenú a doplnenú verziu poznámok z tohto webu. Všetky informácie sú prevzaté zo slajdov, takže by mali byť korektné. Napriek tomu, že som sa poznámky snažil doplniť a skompletizovať, stále som niektoré časti slajdov, ktoré sa mi zdali úplne nepodstatné vynechal, takže pri učení na skúšku odporúčam prejsť aj slajdy – napríklad kvôli obrázkom a konkrétnym príkladom.**

**STOLNÍ SYSTÉMY**

* Desktop – stolní systém
  + Osobní počítač (PC) vyhrazený pro jediného uživatele (v jednom okamžiku)
  + Primární je uživatelské pohodlí
  + Protože uživatel je jediný, mohou být některé bezpečnostní mechanismy vynechány/neimplementovány

**PARALELNÍ SYSTÉMY**

* Úzce vázané systémy
  + Několik vzájemně komunikujících CPU sdílející jednu paměť a hodinový signál
  + Výhody: vyšší propustnost systému, ekonomické využití počítače
* SMP symetrický multiprocesorový systém
  + Všechny procesory jsou si rovné, na všech běží stejná kopie OS
* AMP – asymetrický multiprocesorový systém
  + Každý procesor – specifický úkol např. jeden procesor plánuje ostatním práci, nebo určité typy procesů běží na jednotlivých procesorech

**DISTRIBUOVANÉ SYSTÉMY**

* Volně vázané systémy
  + Každý CPU má vlastní paměť - nekomunikují spolu sdílenou pamětí, ale pomocí komunikačních spojů
  + Výhody: sdílení zdrojů (tiskárny, diskové kapacity), vyšší spolehlivost
  + Architektury
    - Klient-server – řada klientů komunikuje s jedním (nebo více) servery
    - Peer-to-peer sítě – všechny počítače jsou rovnocenné

**REAL-TIME SYSTÉMY**

* RT, real-time systémy, systémy pracující v reálném čase, s pevně stanovenými časovými limity
  + Pro speciální aplikace typu řízení strojů (např. vstřikování v automotoru), sledovací aktivity
  + Hard (přísné) RT systémy
    - při spuštění procesu je stanoven časový limit – OS proces odmítne nebo přijme
    - nutné specializované OS systémy, obvykle bez vnějších pamětí, speciální plánovací algoritmy
  + Soft (tolerantní) RT systémy
    - procesy s vyšší prioritou mají přednost před procesy s nižší prioritou
    - vhodné pro multimedia, robotický průmysl

**KAPESNÍ/MOBILNÍ SYSTÉMY**

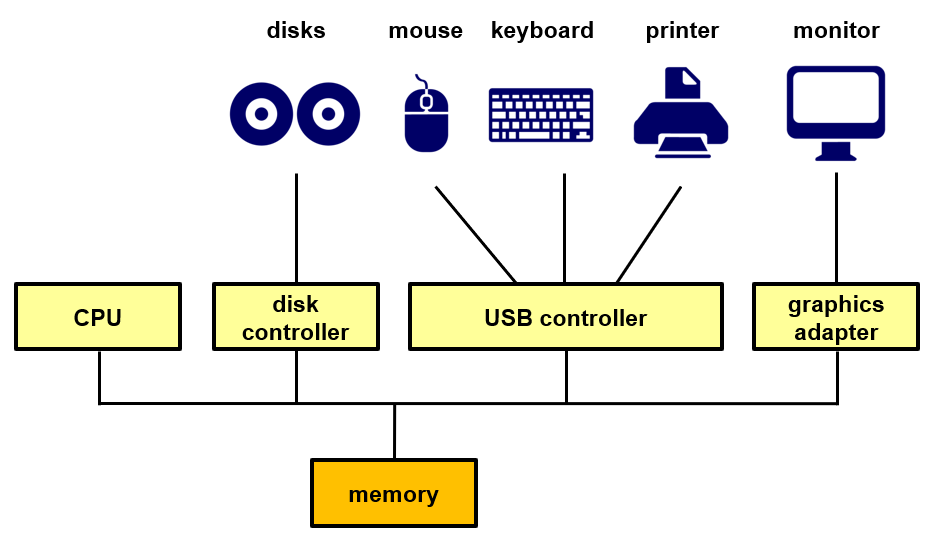
* Palm tops, kapesní systémy, PDA, mobilní telefony, tablety
  + Omezená paměť (volatilní i trvalá), Pomalé procesory, Malé zobrazovací zařízení, Omezená baterie

**MULTIPROGRAMOVÁNÍ**

* Multiprogramování zvyšuje využití CPU
  + přidělování CPU jednotlivým úlohám tak, aby CPU byl využit (téměř) vždy
* V paměti je zároveň několik úloh současně (ne však nutně všechny)
  + plánování úloh (job scheduling) – které úlohy umístit do paměti
  + musíme zajistit ochranu úloh navzájem
* CPU je přidělen úloze, jakmile úloha požádá o I/O operaci, je úloha pozastavena a CPU dostává jiná úloha
  + pro výběr úlohy, která dostane CPU musíme mít CPU plánovací algoritmus
  + jakmile je I/O operace dokončena, je úloha opět přemístěna do fronty úloh připravených ke spuštění
  + CPU je vytížen, dokud mám úlohy, které nečekají na dokončení I/O operací
  + dokud úloha nepožádá o I/O operaci, tak má CPU k dispozici

**MULTITASKING**

* Time-sharing neboli multitasking
  + Logické rozšíření multiprogramování, kdy úloha (dočasně) ztrácí CPU nejen požadavkem I/O operace, ale také vypršením časového limitu
  + CPU je multiplexován, ve skutečnosti vždy běží jen jedna úloha, mezi těmito úlohami se však CPU přepíná, takže uživatelé získají dojem, že úlohy jsou zpracovávány paralelně
* Multitaskingový systém umožňuje řadě uživatelů počítačový systém sdílet
* Uživatelé mají dojem, že počítačový systém je vyhrazen jen pro ně
* Oproti pouhému multiprogramování snižuje dobu odezvy (response time) interaktivních procesů
* Multitaskingové systémy jsou již značně komplexní
  + správa a ochrana paměti, virtuální paměť
  + synchronizace a komunikace procesů
  + CPU plánovací algoritmy, souborové systémy

**Architektúra počítačového systému**

* I/O zařízení typicky mají vlastní vyrovnávací paměť (buffer)
* CPU a I/O zařízení mohou pracovat paralelně
  + např. řadič disku může ukládat data na disk a CPU může něco počítat
  + pokud CPU a I/O zařízení pracují paralelně, měli by se nějak synchronizovat
    - neustálé zjišťování stavu / interrupt

**PROCESOR BEZ PŘERUŠENÍ**

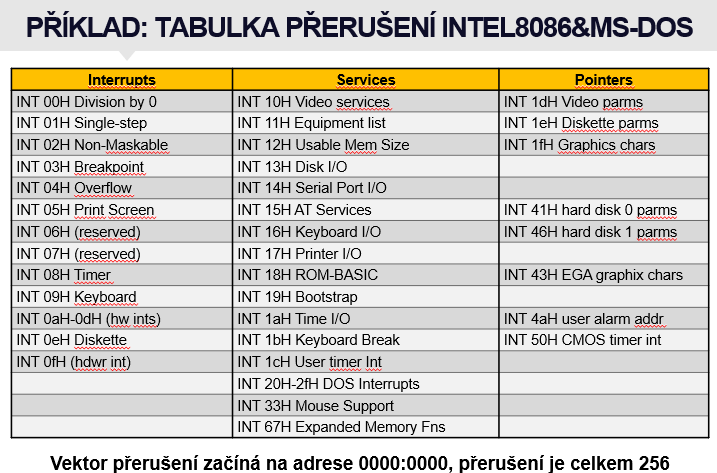
* Neustálý cyklus
  + loop -> získej další instrukci -> proveď instrukci -> end loop
* Pokud chci provést I/O operaci a dozvědět se, že již byla dokončena, musím se periodicky ptát I/O zařízení
  + CPU není v těchto chvílích efektivně využit nebo je programování extrémně náročné

**PROCESOR S PŘERUŠENÍM**

* Základní cyklus je obohacen o kontrolu příznaku přerušení
  + loop -> získej další instrukci -> proveď instrukci -> je-li nastaven příznak přerušení (a přerušení je povoleno), ulož adresu právě prováděného kódu a začni provádět kód obslužné rutiny -> endloop

**PŘERUŠENÍ**

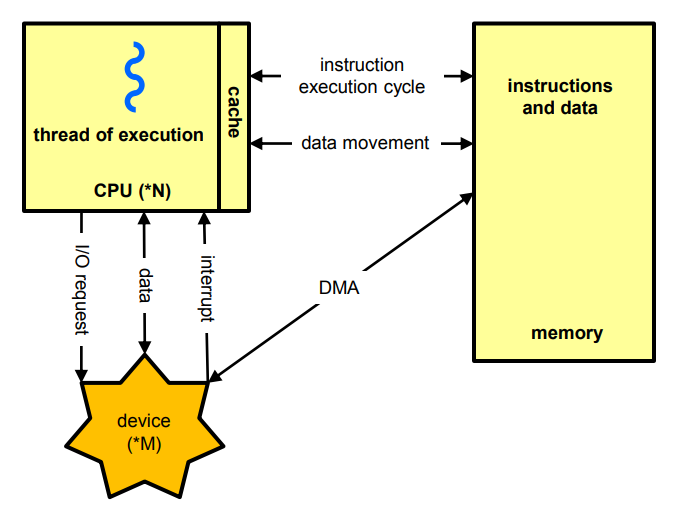
* Přerušení je signál od I/O zařízení, že se stalo něco, co by OS měl zpracovat
* Přerušení přichází asynchronně - OS nemusí aktivně čekat na událost (efektivní využití CPU)
  + při výskytu události se automaticky volá patřičná obslužná rutina
* umístění obslužných rutin pro jednotlivé typy událostí je typicky dáno tabulkou (v paměti) adres rutin (tzv. **vektor přerušení**)
* Aby nedocházelo ke „ztrátě“ přerušení, je při zpracování přerušení další přerušení zakázáno (maskováno)
  + aby se nepřerušovala rutina obsluhující přerušení
  + časově náročnější obslužné rutiny však mohou další přerušení explicitně povolit

****

* Operační systém je řízený přerušeními (interrupt driven)
  + CPU zpracovává uživatelský proces, při příchodu přerušení je spuštěna obslužná rutina OS
  + návratovou adresu ukládá procesor, hodnoty použitých registrů ukládá ovládací rutina
  + Přerušení nemusí být generováno jen HW, přerušení je možné vyvolat i softwarovými prostředky (delenie nulou, špeciálne inštrukcie)

**I/O: DVA PŘÍSTUPY**

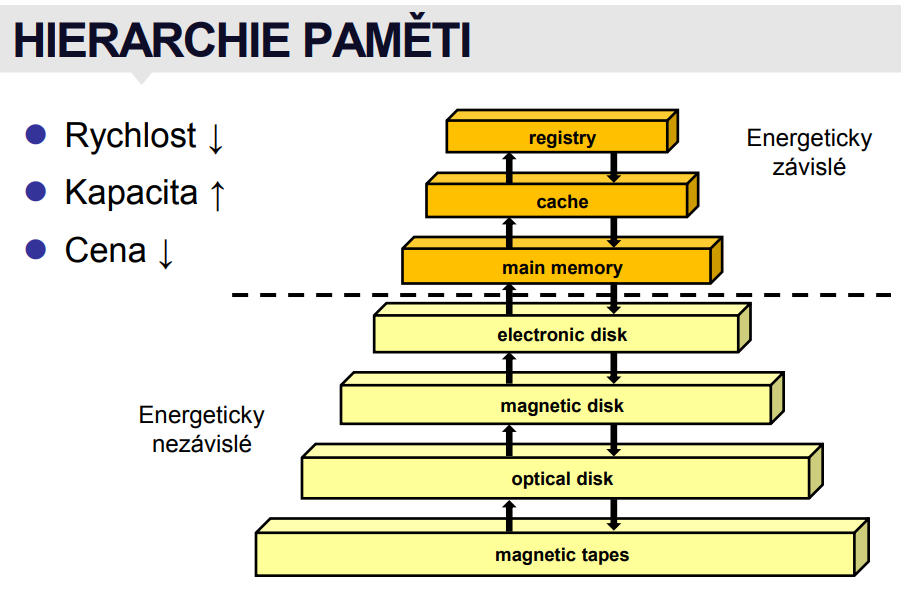
* Synchronní
  + proces požádá o I/O operaci, řízení se uživatelskému procesu vrací až po ukončení I/O operace
  + výhody: jednoduchost, vyhýbáme se souběžnému zpracování I/O požadavků
  + nevýhody: možná neefektivnost
* Asynchronní
  + proces požádá o I/O operaci a řízení se procesu vrací okamžitě – proces môže spustiť ďalšiu I/O operáciu, ktoré potom bežia naraz – synchronný by to nespravil, čakal by, až sa dokončí prvá I/O operácia, až potom by spustil druhú
  + výhody: je možné paralelně pracovat s několika I/O zařízeními
  + nevýhody: komplexnější systém, potřebujeme tabulku stavu I/O zařízení, příkazy pro počkání na dokončení I/O operace apod.

**DMA, Direct Memory Access**

* Způsob jak rychle přenášet data **mezi I/O zařízením a pamětí**
* CPU nemusí přenášet data po jednom bajtu, požádá řadič o přenos celého bloku dat
* CPU požádá o přenos dat a po skončení přenosu dat se o tom dozví pomocí přerušení
* V době přenosu dat soupeří I/O řadič a CPU o přístup do paměti (oba pracují se „stejnou“ pamětí)

**PRIMÁRNÍ PAMĚŤ** (též operační paměť, hlavní paměť)

* jediná větší paměť, kterou může CPU přímo adresovat typicky energeticky závislá (volatilní) současná kapacita: stovky MB až desítky GB, současná rychlost: nanosekundy

**SEKUNDÁRNÍ PAMĚŤ**

* rozšiřuje paměťovou kapacitu počítačového systému
* energeticky nezávislá, vysoká paměťová kapacita (stovky GB až terabajty)
* relativně pomalá doba přístupu (velice variabilní – od mikrosekund až po jednotky sekund)

**KEŠOVÁNÍ, CACHE, MEZIPAMĚŤ**

* Použití rychlejší paměti pro uchování posledně použitých dat z paměti pomalejší (např. cache procesoru vs. operačná paměť alebo operačná vs. disková pamäť)
* pokud jsou data v cachi použijí se tato data, pokud ne, musí se získat z pomalejší paměti, zároveň se přenesou i data okolní (princip časové a prostorové lokálnosti)
* Velikost cache paměti je omezená
* Data se zároveň nacházejí v několika úrovních paměti – problém udržení konzistence

**Ochranné funkcie:**

* Je-li v paměti několik procesů, nechceme, aby se procesy mohly navzájem negativně ovlivňovat
  + přepisování paměti, nespravedlivé získání času CPU alebo souběžný nekoordinovaný přístup k I/O prostředkům
* Proto OS musí tomuto ovlivňování zabránit - často to však nemůže zajistit sám a potřebuje podporu HW

**REŽIMY PROCESORU**

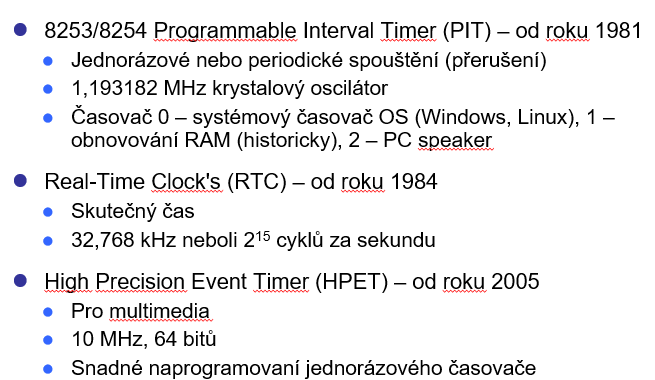
* Běžný způsob ochrany je dvojí režim činnosti procesoru
  + uživatelský režim
  + privilegovaný režim
* Některé instrukce je možné provést jen v privilegovaném režimu
  + např. instrukce pro I/O, nastavování některých registrů (např. některé segmentové registry)
* Z privilegovaného režimu do uživatelského režimu se CPU dostane speciální instrukcí, z uživatelského režimu do privilegovaného režimu se CPU dostává **při zpracování přerušení**

**OCHRANA PAMĚTI**

* Minimálně musíme chránit vektor přerušení a rutiny obsluhy přerušení, jinak by bylo možné získat přístup k privilegovanému režimu procesoru
* Každému procesu vyhradíme jeho paměť, jinou paměť nemůže proces používat
  + ochranu zajišťuje CPU na základě registrů/tabulek nebo principů nastavených OS
  + např. **báze a limit** – proces má přístup jen k adresám v rozsahu báze až báze + limit – jednoduchá implementaćia
  + přístup k nepovoleným adresám způsobí přerušení – to zpracovává OS

**OCHRANA CPU**

* Jak zaručit, že vládu nad procesorem (tj. jaký kód bude CPU vykonávat) bude mít OS?
* Časovač
  + časovač generuje přerušení -> přerušení obsluhuje OS -> ten rozhodne co dál
    - např. odebere jednomu procesu, vybere další připravený proces a ten spustí
  + časovač může generovat přerušení pravidelně nebo je příchod přerušení programovatelný

****

**KOMPONENTY OS**

* Správa procesů
* Správa operační paměti
* Správa souborů
* Správa I/O zařízení
* Správa sekundárních pamětí
* Správa síťových služeb
* Ochranný systém
* Interpret příkazů (shell)

**SPRÁVA PROCESŮ**

* Proces (aktivní entita) je spuštěný program (pasivní entita)
  + jeden program může být zároveň spuštěn řadou uživatelů
  + proces je jednotkou práce systému (procesy jádra OS, uživatelské procesy)
* Proces pro svou činnost potřebuje zdroje
  + procesor, paměť, soubory, alokují se při spuštění nebo při běhu
* Typické aktivity OS v oblasti správy procesů
  + vytváření a ukončování uživatelských a systémových procesů
  + potlačování (suspending) a obnovování (resuming) procesů
  + mechanismy pro synchronizaci procesů
  + mechanismy pro komunikaci mezi procesy
  + mechanismy pro detekci a řešení uváznutí (deadlock)

**SPRÁVA OPERAČNÍ PAMĚTI**

* Při spuštění procesu musíme program nahrát do paměti
* Později může proces vyžadovat dodatečnou paměť pro data a tuto paměť OS vracet
* Jakmile proces končí, musí OS veškerou paměť užívanou procesem opět uvolnit
* Typické aktivity OS v oblasti správy paměti
  + znalost, která část paměti je využívána a kým (kterým procesem)
  + přidělování (alokace) a uvolňování (dealokace) paměti podle požadavků procesů
  + rozhodování o tom, který proces kdy zavést do paměti

**SPRÁVA SOUBORŮ**

* Soubor
  + kolekce souvisejících informací definovaná svým tvůrcem, OS zavádí abstraktní koncept souboru
  + může mít strukturu, ale nemusí
  + z hlediska OS typicky jen posloupnost bajtů
* Typické aktivity OS v oblasti správy souborů
  + Vytváření, rušení či zálohování souborů a adresářů
  + základní operace pro manipulaci se soubory a adresáři (čtení, zápis, seznam souborů v adresáři)

**SPRÁVA I/O ZAŘÍZENÍ**

* Snaha o skrytí specifik jednotlivých I/O zařízení
  + co nejjednotnější přístup k jednotlivých I/O zařízením
  + mnoho OS zpřístupňuje I/O zařízení přes speciální soubory
* Řízení bufferů, kešování, spooling

**SPRÁVA SEKUNDÁRNÍ PAMĚTI**

* Typické sekundární paměti jsou disky
* Správa sekundárních pamětí obvykle formou souborů – souborového systému
* správa volného místa, přidělování místa
* plánování činnosti disku (které požadavky kdy vyřídit)

**SPRÁVA SÍŤOVÝCH SLUŽEB**

* Komunikace je řízena protokolem
* Snaha o transparentnost
  + síťové souborové systémy (SMB, NFS, …)
  + API podobné jako přístup k souborům

**OCHRANNÝ SYSTÉM**

* V multitaskingovém a multiuživatelském OS musíme jednotlivé procesy navzájem chránit
* **Ochrana** je mechanismus, který řídí přístup programů, procesů a uživatelů ke zdrojům počítačového systému
  + HW za pomocí OS zajišťuje, že proces může používat pouze adresy svého adresového prostoru
  + časovač brání jednomu procesu v získání plného přístupu k CPU
  + režim CPU brání uživatelským procesům spouštět privilegované instrukce
* Ochranný mechanismus rozlišuje autorizované a neautorizované použití prostředků

**INTERPRET PŘÍKAZŮ**

* Interpret příkazů, neboli command-line interpreter (command line interface - CLI), neboli shell
  + úkolem je získávat příkazy od uživatele a provádět je
  + Za tento interpret příkazů můžeme považovat i moderní GUI
* Někdy je interpret příkazů přímo součást jádra operačního systému, někdy je to jen speciální program

**INTERNÍ SLUŽBY OS**

* **Alokace zdrojů** - plánovací algoritmy pro přidělování CPU, přidělování přístupu k tiskárnám apod.
* **Účtování** - máme přehled o tom, kteří uživatelé kdy využili které zdroje
* **Ochrana** - autentizace, řízení přístupu

**SYSTÉMOVÁ VOLÁNÍ**

* Systémová volání tvoří rozhraní mezi uživatelským procesem a OS
  + typicky jsou popsána jako instrukce assembleru a jsou uvedena v programátorském manuálu k OS
  + vyšší programovací jazyky obsahují některé funkce, které odpovídají systémovým voláním (např. open, write) a dále knihovní funkce, které poskytují vyšší funkčnost a v rámci této spouští (třeba hned několik) systémových volání (např. fopen, fwrite).
* Různé OS a různé HW platformy mívají různé způsoby jak volat služby OS a různou strukturu těchto služeb
* Existují určité standardy, které usnadňují přenositelnost programového kódu
  + v oblasti UNIXu: **POSIX** (Portable Operating System Interface)
  + V oblasti Windows: Windows API (dříve Win32)
    - Programátorské rozhraní není definováno na úrovni systémových volání jádra
* Způsoby předání parametrů
  + registry, zásobník, blok v paměti <- pointer

**VNITŘNÍ STRUKTURA OS**

* Existuje řada přístupů a implementací
  + jedno velké monolitické jádro
  + modulární, hierarchický přístup
  + malé jádro a samostatné procesy

**Príklad: OS Unix** - sestává ze 2 částí:

* **systémové programy**
* **jádro** - vše, co se nachází pod rozhraním volání systému a nad fyzickým hardware
  + obstarává plnění funkcí z oblastí systému souborů, plánování CPU, správy paměti, …

**HIERARCHICKÁ VRSTVOVÁ ARCHITEKTURA**

* OS se dělí do jistého počtu vrstev (úrovní)
* Každá vrstva je budována na funkcionalitě nižších vrstev
* Nejnižší vrstva (0) je hardware, nejvyšší vrstva je uživatelské rozhraní
* Řeší problém přílišné složitosti velkého systému – dekompozícia na menšie podsystémy
* Nižší vrstva nabízí vyšší vrstvě „primitivní“ funkce (služby) ale nemôže od nej žiadne požadovať
* Jednu vrstvu lze uvnitř modifikovat, aniž to ovlivní ostatní vrstvy – modularita OS
* Nevýhodou je především vyšší režie a tím pomalejší vykonávání systémových volání

**PROVÁDĚNÍ SLUŽEB V KLASICKÉM OS**

* Klasický OS (non-process kernel OS)
  + OS je prováděn jako samostatná entita v privilegovaném režimu
  + procesy – jen uživatelské programy
* Služba se provádí jako součást jádra
* Služba se provádí v rámci procesů
  + obecně lze celý OS provádět v kontextu uživatelského procesu
    - Leží v jeho adresovém prostoru
  + přerušení (volání služby OS) vyvolává implicitně pouze přepnutí režimu procesoru (z uživatelského do privilegovaného), ne změnu kontextu
  + k přepínání kontextu procesů dochází jen tehdy, je-li to nutné z hlediska plánování
  + pro volání procedur v rámci jádra se používá samostatný zásobník
  + program a data OS jsou ve sdíleném adresovém prostoru a sdílí je všechny uživatelské procesy

**STRUKTURA S MIKROJÁDREM -** Microkernel System Structure

* Malé jádro OS plnící pouze několik málo nezbytných funkcí
  + primitivní správa paměti (adresový prostor)
  + komunikace mezi procesy – Interprocess communication (IPC)
* Většina funkcí z jádra se přesouvá do „uživatelské“ oblasti
  + ovladače HW zařízení, služby systému souborů, virtualizace paměti …
  + mezi uživatelskými procesy se komunikuje předáváním zpráv
* Výhody mikrojádra
  + snadná přenositelnost OS, jádro je malé
  + vyšší spolehlivost (moduly mají jasné API a jsou snadněji testovatelné)
  + vyšší bezpečnost (méně kódu OS běží v režimu jádra)
  + flexibilita (jednodušší modifikace, přidání, odebrání modulů)
  + všechny služby jsou poskytovány jednotně (výměnou zpráv)
* Nevýhoda mikrojádra
  + zvýšená režie - volání služeb je nahrazeno výměnou zpráv mezi procesy
* klasický príklad – MACH z 80. rokov
* Windows NT používají hybridní strukturu
  + jádro má vrstevnou strukturu a zajišťuje komunikaci aplikace se „servery“
  + pro jednotlivé typy aplikací (Win32, OS/2, POSIX) existují „servery“ běžící v uživatelském režimu

**MODULARITA LINUX**

* Do linuxového jádra můžeme při běhu přidávat kód – moduly (Loadable Kernel Module – LKM)
* Přesto je Linuxové jádro monolitické – Jde o modularitu kódu jádra ne o modulární architekturu jádra
  + moduly běží stejně jako zbytek jádra v privilegovaném režimu
* LKM umožňují:
  + Přidávat funkčnost za běhu - Např. připojení nového USB zařízení
  + Snižují paměťové nároky jádra
    - Nahráváme jen moduly, které potřebujeme
    - Oproti speciálně zkompilovanému jádru však mají vyšší režii
* /dev/kmem - Možnost přímo číst/měnit paměť jádra za běhu
  + Přístupné pouze pro administrátora, přesto nebezpečné
  + V řadě distribucí už /dev/kmem nenajdeme

**MODULARITA WINDOWS**

* Do jádra Windows můžeme za běhu vkládat ovladače
* Ty běží v privilegovaném režimu jádra
* Podpisy ovládačov - U Windows nejde při podepisování o čistotu jádra či licence
  + Jde především o spolehlivost systému – tj. kvalitu kódu běžícího v privilegovaném režimu a s tím související bezpečnost jádra/systému, DRM (!)
  + User mode drivers – snaha snížit množství kódu běžícího přímo v jádře

**SYSTEM GENERATION (SYSGEN)**

* OS musí být konfigurovatelný na konkrétní sestavu
* Program SYSGEN - získává informace týkající se konkrétní konfigurace konkrétního hardwarového systému
* Bootování - spuštění činnosti počítače zavedením jádra a předáním řízení na vstupní bod jádra pro spuštění
* Bootstrap program **-** uchovávaný v ROM, je schopný nalézt jádro, zavést ho do paměti a spustit ho

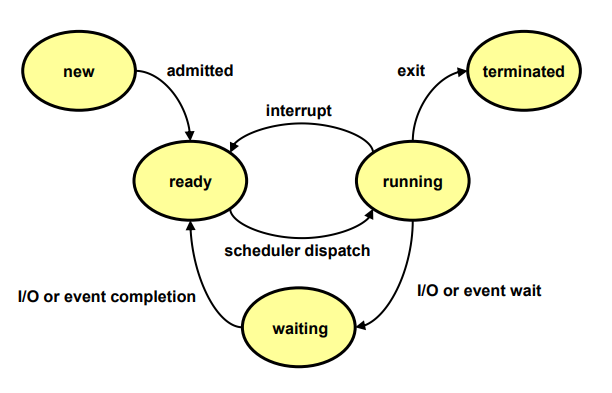
**Bootovanie LINUXU na IBM PC**

* Řídí BIOS - provede se inicializace HW komponent
* na základě uložené konfigurace zjistíme z kterého zařízení se má OS zavést
* Z vybraného zařízení se získá a spustí zavaděč – („boot loader“)
* Fáze 1 - z pevného disku se přečte a spustí prvních 512 bajtů - Master Boot Recordu (MBR)
* Spuštění zavadeče - Ve volitelné fázi 1.5 se zavede kód pro přístup k disku (BIOS nemusí plně umět)
* Ve fázi 2 se zavede zbytek kódu zavaděče.
* Výběr OS a parametrů
* GRUB je schopen pracovat s ext2, ext3, ext4 souborovými systémy
* LILO souborové systémy nezná a pracuje s přímými adresami souborů na disku
* Spuštění jádra - z disku je přečten obraz jádra
  + Tento obraz se dekomprimuje
  + Provede se základní inicializace (např. tabulka stránek paměti)
  + Spustí se jádro (volá se funkce start\_kernel())
* Jádro - Nastaví systém (ovladače přerušení, inicializace zařízení a ovladačů zařízení) a spustí plánovač
  + Vytvoří proces Init (číslo procesu 1) – např. /sbin/init

**SECURE BOOT**

* Snaha o bezpečné zavedení OS
  + Ochrana samotného zavádění komponent OS
  + Prevence rootkitů (bootkitů)
* Systémový firmware skontroluje, či je bootloader podpísaný kľúčom autorizovaným jeho databázou
  + predchádza sa spusteniu nepodpísaného kódu

**PROCES**

* Společné pojmenování pro spuštěný program je proces (task)
* Dále zavádíme pojem vlákno pro „dílčí“ proces v rámci „procesu“
* Proces obsahuje: čítač instrukcí, zásobník, datovou sekci, program (instrukční sekce)
* Hierarchie procesů:
  + rodič, potomek (proces vytvořený na žádost jiného procesu – rodiče)
  + sourozenci (procesy vytvořené jedním rodičem)
* prokládáním běhů procesů maximalizujeme využití

procesoru a minimalizujeme dobu odpovědi

* procesu jsou přidělovány zdroje systému
  + bereme v úvahu priority a vzájemnou výlučnost operací
  + musíme zabránit „uváznutí“ procesů (deadlock)
* **Stavy procesu:**
  + nový (new): právě vytvořený proces
  + běžící (running): některý procesor právě vykonává instrukce procesu
  + čekající (waiting): čeká na určitou událost
  + připravený (ready): čeká na přidělení času procesoru
  + ukončený (terminated): ukončil své provádění
  + odložený čekající a odložený připravený
* **Process Control Block** (PCB) - tabulka obsahující informace potřebné pro definici a správu procesu
  + stav procesu, čítač instrukcí, registry procesoru, informace potřebné pro správu paměti, informace potřebné pro správu I/O, účtovací informace

**PŘEPNUTÍ KONTEXTU**

* Když OS přepojuje CPU z procesu X na proces Y, musí:
  + uchovat (uložit v PCB procesu X) stav původně běžícího procesu
  + zavést stav nově běžícího procesu (z PCB procesu Y)
  + Přepnutí kontextu představuje režijní ztrátu - během přepínání systém nedělá nic efektivního
* Při přerušení musí procesor
  + uchovat čítač instrukcí
  + zavést do čítače instrukcí hodnotou adresy vstupního bodu ovladače přerušení z vektoru přerušení

**FRONTY PLÁNOVÁNÍ PROCESŮ**

* Fronta úloha - seznam všech procesů v systému
* Fronta připravených procesů - seznam procesů uložených v hlavní paměti a připravených k běhu
* Fronta na zařízení - seznam procesů čekajících na I/O operaci
* Fronta odložených procesů - seznam procesů čekajících na přidělení místa v hlavní paměti
* Fronta na semafor - seznam procesů čekajících na synchronizační událost

**STRATEGICKÝ (dlhodobý) PLÁNOVAČ**

* Obecně nemusím mít všechny úlohy, které chci spustit, v operační paměti **-** fronta všech úloh může být značně dlouhá a plánovač musí rozhodnout, které úlohy zavést do paměti a spustit
* Toto je úkol dlouhodobého (strategického) plánovače
  + vybírá který proces lze zařadit mezi připravené procesy
  + plánovač je spouštěn je relativně málo často – typicky při ukončení jednoho procesu rozhodne, kterou úlohu dále vybrat k zavedení do paměti a spuštění
  + nemusí být super rychlý

**KRÁTKODOBÝ PLÁNOVAČ**

* Přiděluje procesor připraveným procesům - vybírá jeden z procesů, které jsou zavedeny operační paměti a které jsou „připravené“
* Je spúšťaný často, preto musí byť rýchly
* Plánovací rozhodnutí může vydat v okamžiku, kdy proces:
  1. přechází ze stavu běžící do stavu čekající
  2. přechází ze stavu běžící do stavu připravený
  3. přechází ze stavu čekající do stavu připravený
  4. končí
* Případy 1 a 4 se označují jako nepreemptivní plánování (plánování bez předbíhání)
* Případy 2 a 3 se označují jako preemptivní plánování (plánování s předbíháním)
* Je spouštěn často (např. každých 10ms), proto musí být rychlý
* **Dispečer** - Výstupní modul krátkodobého plánovače nebo plánovač sám, který předává procesor procesu vybranému krátkodobým plánovačem
  + Předání zahrnuje:
    - přepnutí kontextu
    - přepnutí režimu procesoru na uživatelský režim
    - skok na odpovídající místo v uživatelském programu pro opětovné pokračování procesu
  + **Dispečerské zpoždění** (Dispatch latency) - Doba, kterou potřebuje dispečer pro pozastavení běhu jednoho procesu a start běhu jiného procesu
* Kritéria plánovania:
  + Využití CPU - cílem je udržení CPU v kontinuální užitečné činnosti
  + Propustnost - počet procesů, které dokončí svůj běh za jednotku času
  + Doba obrátky - doba potřebná pro provedení konkrétního procesu
  + Doba čekání - doba, po kterou proces čekal ve frontě „připravených“ procesů
  + Doba odpovědi - doba, která uplyne od okamžiku zadání požadavku do doby první reakce (první odpovědi, nikoli poskytnutí plného výstupu)

**ODKLÁDÁNÍ PROCESŮ**

* Každý proces musíme jednou zavést do RAM alespoň částečně (část může zůstat na disku a dohrána on-demand – viz některé optimalizace)
* Příliš mnoho procesů v RAM však snižuje výkonnost i při využití virtuální paměti
* OS musí provádění některých procesů odložit -- „vrátit“ na disk
* Dva nové stavy: odložený čekající a odložený připravený

**STŘEDNĚDOBÝ (taktický) PLÁNOVAČ**

* vybírá který proces lze zařadit mezi odložené procesy
* vybírá který odložený proces lze zařadit mezi připravené procesy
* Náleží částečně i do správy operační paměti
  + kapacita FAP je omezená a odložené procesy uvolňují paměť

**Vytvoření procesu**

* Rodič vytváří potomky (další procesy)
  + Potomci mohou vytvářet další potomky - vzniká strom procesů
* Sdílení zdrojů – varianty při vytváření potomků
  + rodič a potomek sdílejí zdroje původně vlastněné rodičem
  + potomek sdílí rodičem vyčleněnou podmnožinu zdrojů s rodičem
  + potomek a rodič jsou plně samostatné procesy, nesdílí žádný zdroj
* Běh
  + rodič a potomek mohou běžet souběžně
  + rodič čeká na ukončení potomka

**Ukončení procesu**

* Proces provede poslední příkaz a sám požádá OS o ukončení
  + výstupní data procesu se předají rodiči (pokud o to má zájem)
  + zdroje končícího procesu se uvolňují operačním systémem
* O ukončení procesu žádá jeho rodič (nebo jiný proces s dostatečnými právy nebo OS), protože např.:
  + potomek překročil stanovenou kvótu přidělených zdrojů
  + úkol přidělený potomkovi rodič již dále nepotřebuje
  + rodič končí svoji existenci a nebylo povoleno, aby potomek přežil svého rodiče
  + může docházet ke kaskádnímu ukončování (ukončí se celá větev stromu procesů)

**KOOPERUJÍCÍ PROCESY**

* Nezávislé procesy - nemohou se vzájemně ovlivňovat
* Kooperující procesy - mohou ovlivňovat běh jiných procesů nebo jiné procesy mohou ovlivňovat jejich běh
* Přínosy kooperace procesů: sdílení informací, urychlení výpočtů, modularita, pohodlí při programování
* Příklady typových úloh kooperace: producent – konzument, klient – server

**Linux a procesy**

* OS s preemptivním multitaskingem
* Služby OS :
  + fork – vytvoří nový proces jako kopii rodiče
  + execve – nahradí současný proces spuštěním jiného programu
  + exit (resp. \_exit) – ukončí proces
  + wait, waitpid – čeká na ukončení potomka

**VLÁKNA**

* Každé vlákno si udržuje svůj vlastní
  + zásobník, PC (program counter), registry, TCB (Thread Context Block)
* Vlákno může přistupovat k paměti a ostatním zdrojům svého procesu
  + vlákna jednoho procesu zdieľajú všetky zdroje procesu
  + jakmile jedno vlákno změní obsah buňky, všechny ostatní vlákna to vidí
  + soubor otevřený jedním vláknem mají k dispozici všechny ostatní vlákna
* Jednovláknový OS nepodporuje koncept vláken (nezná pojem vlákno) – MS-DOS, UNIX
* Multivláknový OS podporuje koncept více vláken v rámci procesů – Windows, Solaris
* Výhody
  + vlákno se vytvoří rychleji než proces
  + vlákno se ukončí rychleji než proces
  + mezi vlákny se rychleji přepíná než mezi procesy
  + jednodušší programování (jednodušší struktura programu)
  + využití multiprocesorových strojů (vlákna jednoho procesu mohou běžet na různých CPU)
  + když vlákno čeká na ukončení I/O operace, může běžet jiné vlákno téhož procesu, aniž by se přepínalo mezi procesy (což je časově náročné)
  + vlákna jednoho procesu sdílí paměť a deskriptory otevřených souborů a mohou mezi sebou komunikovat, aniž by k tomu potřebovaly služby jádra (což by bylo pomalejší)
* Příklady:
  + síťový souborový server musí vyřizovat řadu požadavků klientů, pro vyřízení každého požadavku vytváří samostatné vlákno (efektivnější než samostatný proces)
  + 1 vlákno zobrazuje menu a čte vstup od uživatele a současně 1 vlákno provádí příkazy uživatele, překreslování obrazovky souběžně se zpracováním dat
* Konzistence
  + vlákna jedné aplikace sa vykonávajú paralelne, proto se musí mezi sebou synchronizovat, aby se zachovala konzistentnost dat (musíme zabránit současné modifikaci stejných dat dvěmi vlákny...)
* Tři klíčové stavy vláken: běží, připravený, čekající
* Vlákna se (samostatně) neodkládají - všechny vlákna jednoho procesu sdílejí stejný adresový prostor
* Ukončení procesu ukončuje všechny vlákna existující v rámci tohoto procesu

**VLÁKNA NA UŽIVATELSKÉ ÚROVNI** (User Level Threads – ULT)

* Správa vláken se provádí prostřednictvím vláknové knihovny („thread library“) na úrovni uživatelského / aplikačního programu
* Jádro o jejich existenci neví
  + přepojování mezi vlákny nepožaduje provádění funkcí jádra
  + nepřepíná se ani kontext procesu ani režim procesoru
* Plánování přepínání vláken je specifické pro konkrétní aplikaci
  + aplikace si volí pro sebe nejvhodnější (např. plánovací) algoritmus
* Threads library“ obsahuje funkce pro
  + vytváření a rušení vláken
  + předávání zpráv a dat mezi vlákny
  + plánování běhů vláken
  + uchovávání a obnova kontextů vláken
* Co dělá jádro pro vlákna na uživatelské úrovni
  + jádro neví o aktivitě vláken, proto manipuluje s celými procesy
  + když některé vlákno zavolá službu jádra, je blokován celý proces dokud se služba nesplní
  + pro „thread library“ je takové vlákno ale stále ve stavu „běží“
  + stavy vláken jsou na stavech procesu nezávislé
* Výhody
  + přepojování mezi vlákny nepožaduje provádění jádra - nepřepíná se ani kontext ani režim procesoru
  + plánování je specifické pro konkrétní aplikaci - aplikace volí si pro sebe nejvhodnější algoritmus
  + ULT mohou běžet pod kterýmkoliv OS - není vyžadována podpora na úrovní jádra OS
  + ULT potřebují uživatelskou knihovnu (ke slinkovaní s aplikací)
* Nevýhody
  + většina volání služeb OS způsobí blokování celého procesu (tj. všech vláken procesu)
  + jádro může přidělovat procesor pouze procesům, dvě vlákna stejného procesu nemohou běžet na dvou procesorech

**VLÁKNA NA ÚROVNI JÁDRA** (Kernel-Level Threads – KLT)

* Správu vláken podporuje jádro, nepoužívá se „thread library“
  + používá se API pro vláknové služby jádra
  + informaci o kontextu procesů a vláken udržuje jádro
  + přepojování mezi vlákny aktivuje jádro, plánování na bázi vláken již v jádře OS
* Výhody
  + jádro může současně plánovat běh více vláken stejného procesu na více procesorech
  + k blokování dochází na úrovni vlákna (není blokován celý proces)
  + I programy jádra mohou mít multivláknový charakter
* Nevýhody
  + přepojování mezi vlákny stejného procesu zprostředkovává jádro (tj. pomaleji)
  + při přepnutí vlákna se 2x přepíná režim procesoru (tj. režie navíc)

**KOMBINACE VLÁKEN ULT/KLT**

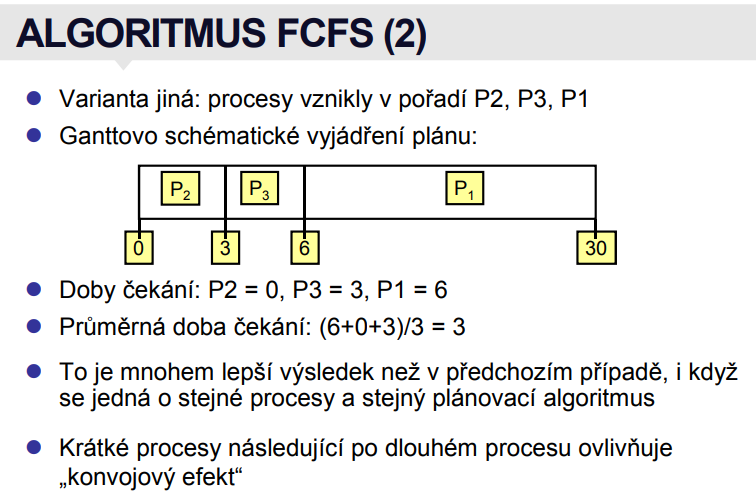
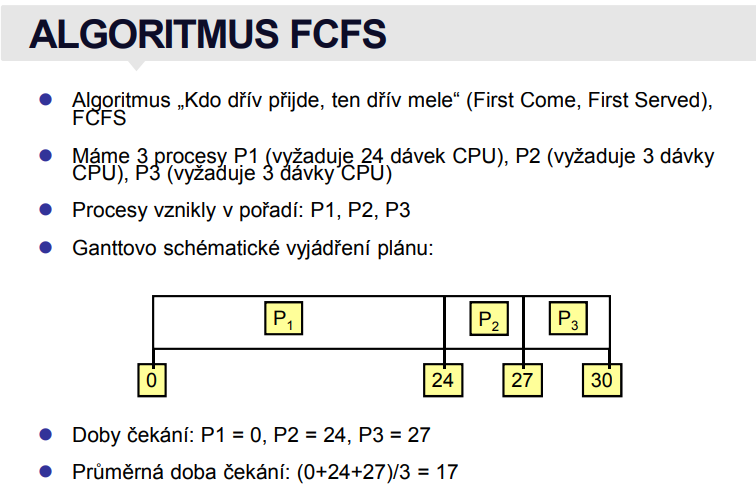
* Vlákna se vytvářejí v uživatelském prostoru
* Většina plánování a synchronizace se dělá v uživatelském prostoru
* Programátor může nastavit počet vláken na úrovni jádra
* Lze kombinovat přínosy oboru přístupů
* Multivláknové modely:
  + n : 1 - více ULT se zobrazuje do 1 KLT
  + 1 : 1 - každý ULT se zobrazuje do 1 KLT
  + n : m - více ULT se může zobrazovat do více KLT

Vlákna na LINUX – UNIX – POSIX – knihovna „pthreads“

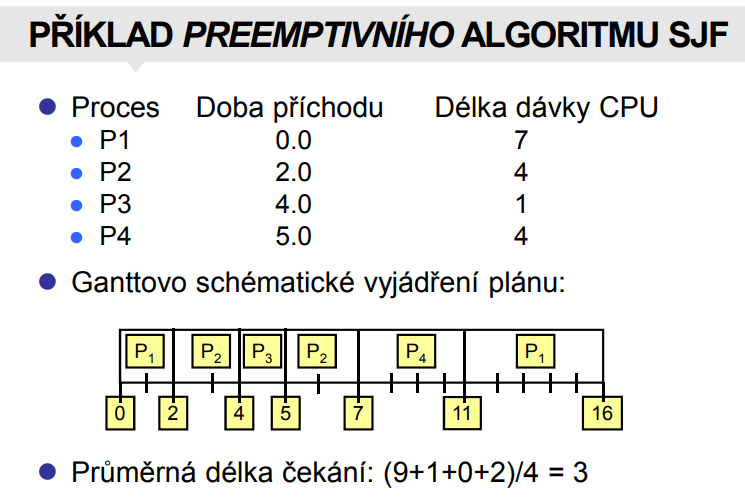
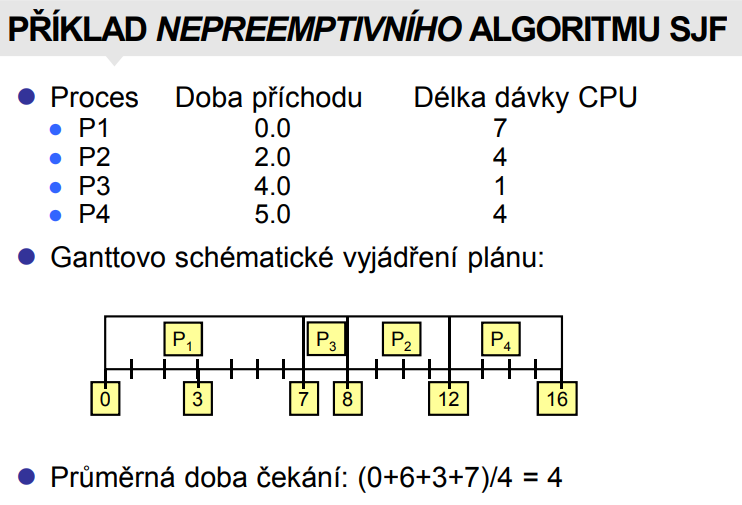
* služby: pthread\_create, pthread\_exit, pthread\_join, pthread\_detach, pthread\_attr\_init
* Implementace POSIX threads:
  + LinuxThreads - odpovídá POSIX standardu IEEE 1003.1c až na ovladače signálů.
    - Vlákna mají různá PID (Process Identifier)
    - Není nadále vyvíjeno
  + NTPL (Native POSIX Threads Library for Linux)
    - Nahradilo LinuxThreads, lepší výkon, vyžaduje jádro řady 2.6
    - Dnes součást knihovny GNU C
  + NGPT (Next Generation POSIX Threads) - alternativa k NTPL, které se neprosadila

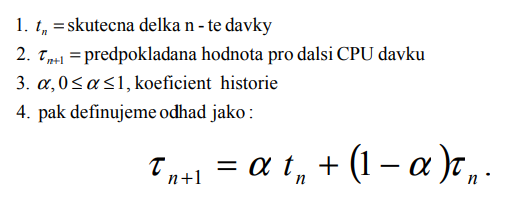
**ALGORITMUS FCFS (**First Come, First Served (basically FIFO))

* procesy sú obslúžené v poradí, v akom prišli
* efektivita závisí na dĺžke a poradí procesov (ak sú na začiatku fronty krátke procesy je to efektívne, ak sú dlhé tak to nie je efektívne)



**ALGORITMUS SJF (**Shortest Job First)

* Musíme znát délku příštího požadavku na dávku CPU pro každý proces
* Vybírá se proces s nejkratším požadavkem na CPU
* Dvě varianty:
  + nepreemptivní, bez předbíhání
    - jakmile se CPU předá vybranému procesu, tento nemůže být předběhnut žádným jiným procesem, dokud přidělenou dávku CPU nedokončí
  + preemptivní, s předbíháním
    - jakmile se ve frontě připravených procesů objeví proces s délkou dávky CPU kratší než je doba zbývající k dokončení dávky právě běžícího procesu, je právě běžící proces ve využívání CPU předběhnut novým procesem
    - tato varianta se rovněž nazývá Shortest-Remaining-Time-First (SRTF)
* SJF je optimální algoritmus (pro danou množinu procesů dává minimální průměrnou dobu čekání)   
  



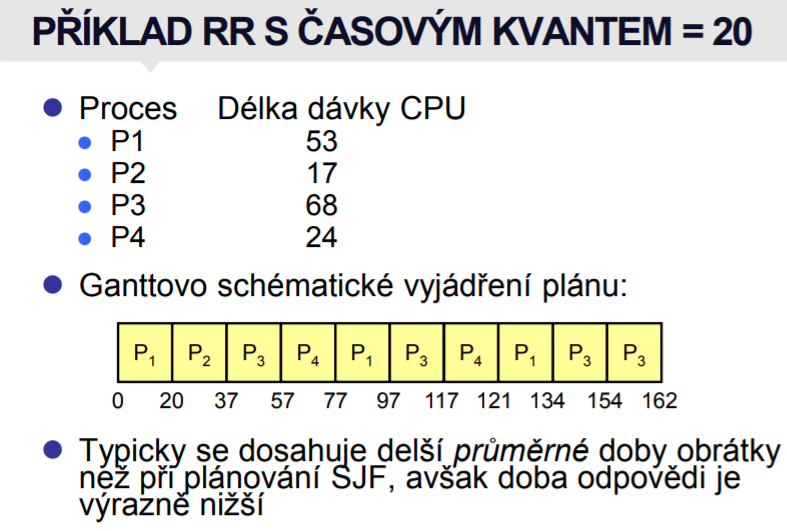
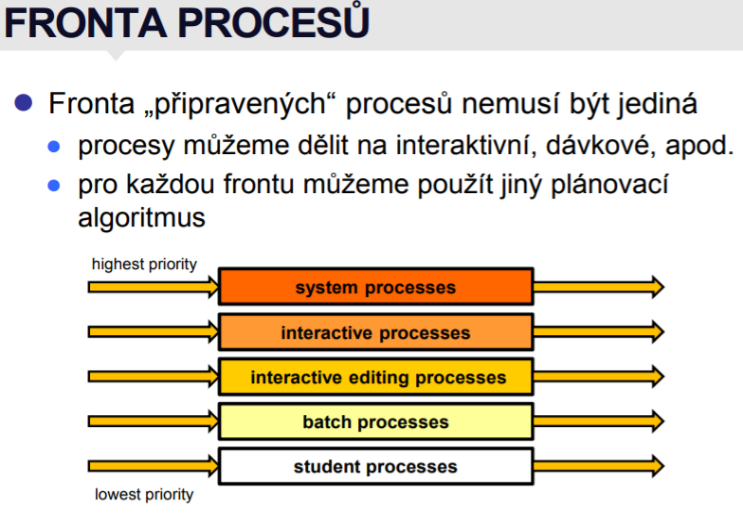
**URČENÍ DÉLKY PŘÍŠTÍ DÁVKY CPU PROCESU**

* Délku příští dávky CPU procesu neznáme, můžeme ji pouze odhadovat
* To můžeme udělat na základě historie
* Musíme znát délky předchozích dávek CPU
* Použijeme exponenciální průměrování:

**PRIORITNÍ PLÁNOVÁNÍ**

* S každým procesem je spojeno prioritní číslo
  + prioritní číslo – preference procesu pro výběr příště běžícího procesu
  + CPU se přiděluje procesu s největší prioritou
  + nejvyšší prioritě obvykle odpovídá nejnižší prioritní číslo
* Opět dvě varianty
  + nepreemptivní, bez předbíhání
    - jakmile proces získá přístup k CPU nemůže být předběhnut jiným procesem dokud dávku neukončí
  + preemptivní, s předbíháním
    - jakmile se ve frontě připravených procesů objeví proces s vyšší prioritou než je priorita běžícího procesu, je běžící proces předběhnut
* SJF je prioritní plánování, prioritou je předpokládaná délka příští CPU dávky
* stárnutí - procesy s nižší prioritou se nemusí nikdy provést
  + řešení: zrání – priorita se s postupem času zvyšuje

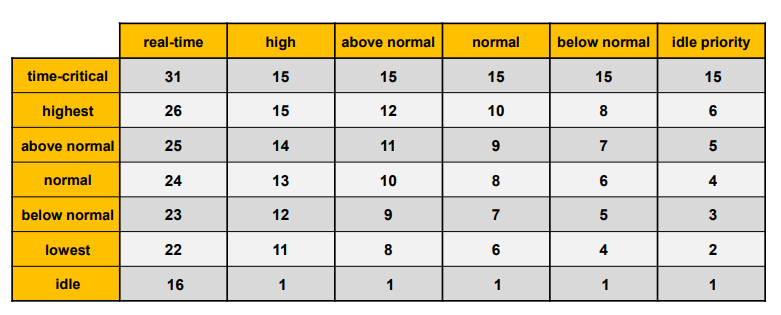
**ROUND ROBIN (RR)**

* Každý proces dostává CPU na malou jednotku času – časové kvantum **-** desítky až stovky ms
* Po uplynutí této doby je běžící proces předběhnut nejstarším procesem ve frontě připravených procesů a zařazuje se na konec této fronty
* Je-li ve frontě připravených procesů n procesů a časové kvantum je q, pak každý proces získává 1/n doby CPU, najednou nejvýše q časových jednotek
* Žádný proces nečeká na přidělení CPU déle než (n-1)q časových jednotek
* Výkonnostní hodnocení
  + q velké → ekvivalent FIFO
  + ****q malé → velká režie; v praxi musí být q dostatečně velké s ohledem na režii přepínání kontextu

**PLÁNOVACÍ ALGORITMUS LINUX (súčasť jadra OS)**

* Skládá se ze 2 funkcí
  + schedule() – plánování procesů
  + do\_timer() – aktualizuje informace o procesech
* Časové kvantum je 1/100 sekundy
* 4 kategorie procesů z hlediska plánování:
  + SCHED\_FIFO – sú plánované metódou FIFO
  + SCHED\_RR – sú plánované metódou RR
  + SCHED\_OTHER – všetky klasické timesharingové procesy
  + SCHED\_BATCH
* první dva typy jsou procesy se zvláštními nároky na plánování (soft real time), mají přednost před ostatními procesy a může je vytvářet pouze root
* procesy SCHED\_FIFO a SCHED\_RR mají přirazenu prioritu (1-99), při plánování jsou vybírány procesy s vyšší prioritou, plánovací algoritmu je preemptívny
* Plánovací algoritmus:
  + Do jádra 2.4 algoritmus procházející globální (všechny procesory) frontu a hledající vhodný proces
  + Od jádra 2.6 (resp 2.5) nový, tzv. O(1) algoritmus, fronty procesů per-CPU
  + Od 2.6.23 nahrazen algoritmem CFS (completely fair scheduler), který pro seznamy procesů používá červeno-černý strom. Výběr procesu pro běh na procesoru v konstantním čase, jeho znovuvložení

**PLÁNOVACÍ ALGORITMUS WIN32**

* procesy při vytvoření přiděleny do jedné z následujících tříd
  + Idle
  + Below Normal
  + Normal
  + Above Normal
  + High
  + Realtime
* Vlákna dále mají relativní prioritu v rámci třídy, do které patří
  + Idle
  + Lowest
  + Below\_Normal
  + Normal
  + Above\_Normal
  + Highest
  + Time\_Critical
* Plánuje vlákna, nie procesy, priorita 0 až 31
* 32 front (FIFO seznamů) vláken, která jsou „připravena“
  + pro každou úroveň priority jedna fronta
  + fronty jsou společné pro všechny procesory
  + když je vlákno „připraveno“ buď běží okamžitě, nebo je umístěno na konec fronty „připravených“ procesů ve své prioritě
* na jednoprocesorovém stroji vždy běží vlákno s nejvyšší prioritou
* V rámci jedné prioritní skupiny se plánuje algoritmem round-robin pomocí časových kvant

**SPRÁVA PAMĚTI**

* Pro běh procesu je nutné, aby program, který je vykonáván byl umístěn v operační paměti
* Z programu se stává proces (aktivní entita schopná spuštění na CPU) provedením celé řady kroků
  + naplnění tabulek, umístění do operační paměti
  + vázání adres instrukcí a dat na adresy operační paměti
* Program je složen z částí se vzájemně odlišnými vlastnostmi
  + moduly s instrukcemi jsou označovány „jen ke spuštění“ (execute-only)
  + datové moduly jsou buďto „read-only“ nebo „read/write“
  + některé moduly jsou „soukromé“ (private), jiné jsou veřejné „public“
* Více procesů může sdílet společnou část paměti, aniž by se tím porušovala ochrana paměti

**VÁZÁNÍ ADRES**

* Při kompilaci
  + umístění v paměti je známé apriori, lze generovat absolutní kód
  + při změně umístění se musí program znovu přeložit
* Při zavádění
  + umístění v paměti není známé v době kompilace, generuje se přemístitelný kód (relocatable code)
* Za běhu
  + jestliže proces může měnit svoji polohu během provádění, vázání se zpožďuje na dobu běhu
  + musí být dostupná hardwarová podpora - bázové registry, mezní registry, …

**MEMORY-MANAGEMENT UNIT** (MMU)

* Hardwarový modul převádějící logické adresy na fyzické adresy
* Uživatelský program pracuje s logickými adresami, uživatelský program nevidí fyzické adresy
* Připočítává se obsah „relokačního registru“ k adresám generovaným uživatelským procesem v okamžiku, kdy je předávána jako ukazatel do operační paměti

**ADRESOVÝ PROSTOR**

* Logický adresový prostor (LAP), fyzický adresový prostor (FAP)
  + LAP – (logická adresa, virtuální adresa) dána adresou ve strojovém jazyku, generuje CPU
  + FAP – (fyzická) adresa akceptovaná operační pamětí
* Logické a fyzické adresové prostory se shodují v době kompilace a v době zavádění
* Logické a fyzické adresové prostory mohou být rozdílné při vázání v době běhu

**DYNAMICKÉ ZAVÁDĚNÍ** (dynamic loading)

* Programový kód se nezavádí dokud není zavolán (spuštěn)
* Dosahuje se lepšího využití paměti - nepoužívané části kódu se nikdy nezavádí
* Užitečná technika v případech, kdy se musí velkými programovými moduly řešit zřídka se vyskytující alternativy - Např. tzv. plug-in moduly
* Program nevyžaduje žádnou speciální podporu od operačního systému
* Linux – dlopen, dlsym, dlclose
* Windows API – LoadLibrary, GetProcAddress, FreeLibrary

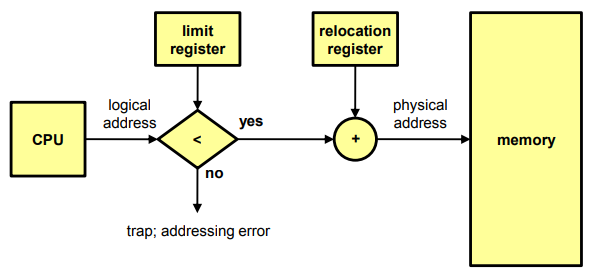
**DYNAMICKÉ VÁZÁNÍ** (dynamic linking)

* Vázání je odkládáno na dobu běhu
* Pro umístění příslušných knihovních programů rezidentních v operační paměti se používá kód malého rozsahu – stub
* Při zavolání stub nahradí sám sebe adresou skutečné funkce a předá jí řízení
* OS musí kontrolovat, zda funkce je mapována do paměti procesu
* Je to technika vhodná zvláště pro knihovní funkce

**PŘEKRYVY, OVERLAYS**

* V operační paměti se uchovávají pouze ty instrukce a data, která jsou potřeba po celou dobu běhu
* Technika, která je nutná v případech, kdy je přidělený prostor paměti menší než je souhrn potřeb procesu
* Vyvolávání překryvů je implementované programátorem, od OS se nepožaduje žádná speciální podpora
* Návrh překryvové struktury ze strany programátora je značně složitý

**SOUVISLÉ OBLASTI**

* Operační paměť se dělí do dvou sekcí
  + rezidentní OS, obvykle na počátku FAP s tabulkou ovladačů přerušení
  + uživatelské procesy
* ****Přidělování jedné souvislé části paměti
  + pro ochranu procesů uživatelů mezi sebou a OS lze použít schéma s relokačním registrem
  + **relokační registr**: hodnota nejmenší fyzické adresy paměti procesu
  + **mezní registr**: rozpětí logických adres, logická adresa musí být menší nebo rovna meznímu registru
* Přidělování několika částí paměti
  + díra – blok dostupné paměti **-** bloky jsou roztroušeny po FAP
  + evidenci o přidělených a volných sekcí udržuje OS

**PŘIDĚLOVÁNÍ PAMĚTI**

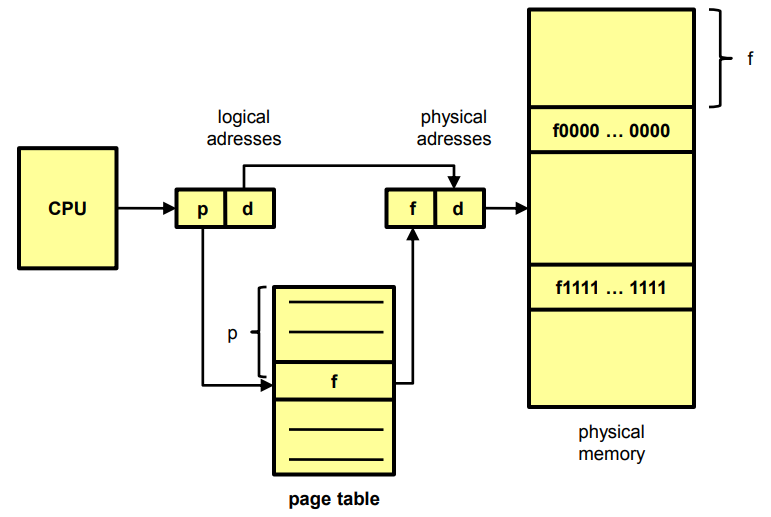
* Kterou oblast délky n přidělit, když volná paměť je rozmístěna ve více souvislých nesousedních sekcích?
* First-fit - přiděluje se první dostatečně dlouhá volná oblast resp. její počátek
* Best-fit - přiděluje se nejmenší dostatečně dlouhá volná oblast resp. její počátek
  + generují se velmi malé (nejmenší) možné volné díry
* Worst-fit - přiděluje se největší dostatečně dlouhá volná oblast resp. její počátek
  + generují se největší možné volné díry
* Z hlediska rychlosti a kvality využití paměti jsou First-fit a Best-fit jsou lepší techniky než technika Worst-fit

**PROBLÉM FRAGMENTACE**

* Vnější fragmentace - souhrn volné paměti je dostatečný, ale ne v dostatečné souvislé oblasti
* Vnitřní fragmentace ­- přidělená paměť je větší než požadovaná velikost, část přidělené paměti je nevyužitá
* Snižování vnější fragmentace setřásáním
  + přesouvají se obsahy paměti s cílem vytvořit (jeden) velký volný blok
  + použitelné jen když je možná dynamická relokace (viz MMU), provádí se v době běhu
  + problém I/O – s vyrovnávacími paměťmi plněnými z periférií autonomně nelze hýbat, umisťují se proto do prostoru OS

**STRÁNKOVÁNÍ**

* LAP procesu nemusí být jedinou souvislou sekcí FAP, LAP se zobrazuje do (po částech volných) sekcí FAP
* FAP se dělí na sekce zvané rámce (frames)
  + pevná délka, délka v násobcích mocnin 2 (obvykle mezi 512 až 8192 bajty)
* LAP se dělí na sekce zvané stránky (pages)
  + pevná délka, shodná s délkou rámců
* Udržujeme seznam volných rámců
* Program délky n stránek se umístí (zavede) do n rámců
* logická adresa → fyzická adresa − pomocí překladové tabulky nastavované OS a interpretované MMU
* Vzniká vnitřní fragmentace, neboť paměť je procesu přidělována v násobcích velikosti rámce



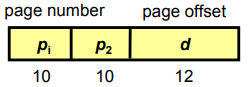
**DYNAMICKÝ PŘEKLAD ADRES**

* Logická adresa (generovaná CPU) se dělí
  + číslo stránky, p
    - index do tabulky stránek
    - index bázové adresy rámce přiděleného stránce, které patří logická adresa
  + offset ve stránce, d
    - přičítáme k začátku stránky/rámce

**TABULKA STRÁNEK**

* Je uložena v operační paměti
* Její počátek a konec je odkazován registrem
  + Page-table base register (PTBR), Page-table length register (PTLR)
* Zpřístupnění údaje / instrukce v operační paměti, vyžaduje dva přístupy do operační paměti
  + jednou do tabulky stránek
  + jednou pro údaj/instrukci
* Problém zhoršení efektivnosti dvojím přístupem lze řešit speciální rychlou hardwarovou cache pamětí
  + asociativní paměť, translation look-aside buffers (TLB)

**DVOUÚROVŇOVÁ TABULKA STRÁNEK**

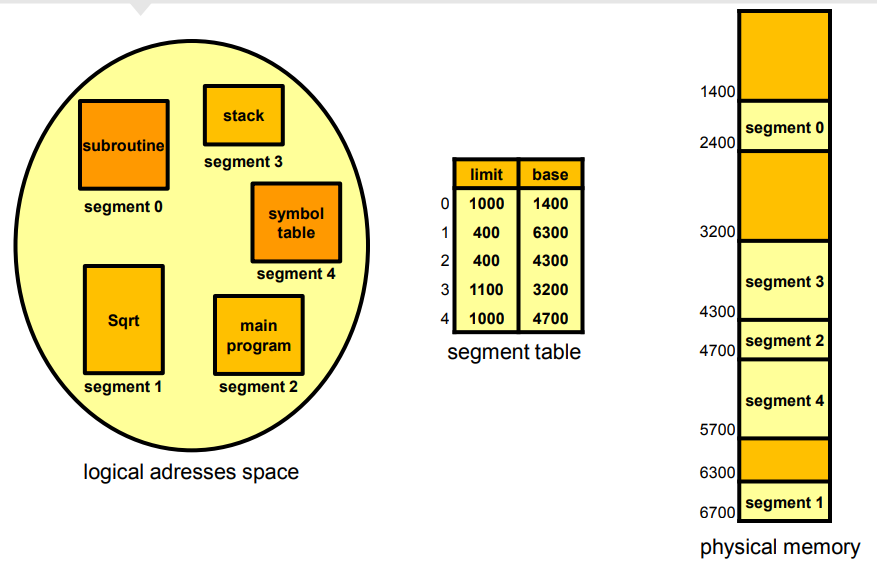
* 32-bitový procesor s 4KB stránkou
* PT (tabulka stránek) je stránkovaná
* Logická adresa
  + číslo stránky: 20 bitů
  + adresa ve stránce: 12 bitů
* Číslo stránky se dále dělí
  + číslo stránky 10-bitů
  + adresa v tabulce stránek 10-bitů

**INVERTOVANÁ TABULKA STRÁNEK**

* 1 položka v PT pro každý rámec paměti
* Obsahuje
  + logickou (virtuální) adresu stránky uchovávané v odpovídajícím rámci
  + informaci o procesu, který stránku vlastní
* Snižuje se velikost paměti potřebné pro uchovávání PT
* Zvyšuje se doba přístupu do PT, indexový přístup je nahrazen prohledáváním
  + lze zvýšit efektivnost využitím hašování
    - perfektní hašování – 1 přístup k PT
    - jinak několik přístupů k PT

**SDÍLENÍ STRÁNEK**

* Sdílený kód je umístěn v paměti pouze jednou
  + kód je reentrantní a „read-only“
  + musí se nacházet na stejné logické adrese ve všech procesech
* Privátní data
  + každý proces udržuje svoji privátní kopii dat (a nesdíleného kódu)

**SEGMENTOVÁNÍ**

* Logická adresa je dvojice (segment s, offset d)
* Tabulka segmentů, Segment table, ST
  + base – počáteční adresa umístění segmentu ve FAP
  + limit – délka segmentu
* Segment-table base register (STBR)
  + odkaz na umístění ST v paměti
* Segment-table length register (STLR)
  + počet segmentů, s je legální když s < STLR
* Relokace – dynamická, pomocí ST

**VIRTUÁLNÍ PAMĚŤ**

* Uložení obrazu LAP (virtuální paměti) v externí paměti
  + nepoužívá se standardní systém souborů OS, používají se metody optimalizované pro tento účel
  + speciální partition/slice disku
* Stránkování / segmentaci musí podporovat hardware správy paměti
  + stránkování na žádost, segmentování na žádost
  + žádost – dynamicky kontextově generovaná indikace nedostatku
* OS musí být schopen organizovat tok stránek / segmentů mezi vnitřní a vnější pamětí
* ve FAP se mohou nacházet pouze části programů nutné pro bezprostřední řízení procesů
* LAP může být větší než FAP, adresové prostory lze sdílet mezi jednotlivými procesy
  + lze efektivněji vytvářet procesy
* Techniky implementace
  + Stránkování na žádost, Demand Paging
  + Segmentování na žádost, Demand Segmentation
* Část procesu umístěnou ve FAP nazýváme „rezidentní množinou“ (resident set)
* Odkaz mimo rezidentní množinu způsobuje přerušení výpadek stránky (page fault)
* OS označí proces za „čekající“, spustí I/O operace a provede nutnou správu paměti pro zavedení odkazované části do FAP, mezitím běží jiný proces (jiné procesy)
* Po zavedení odkazované části se generuje I/O přerušení - OS příslušný proces umístí mezi připravené procesy
* Překlad adresy LAP na adresu FAP se dělá indexováním tabulky PT/ST pomocí hardware CPU
* Ve FAP lze udržovat více procesů najednou
  + čím více je procesů ve FAP, tím je větší pravděpodobnost, že stále bude alespoň jeden připravený
* Lze realizovat procesy požadující více paměti než umožňuje FAP
* Stránkování na žádost (Demand paging)
  + stránka se zobrazuje do FAP při odkazu na ni, pokud ve FAP není již zobrazena
  + počáteční shluky výpadků stránek
* Předstránkování (Prepaging)
  + umístění na vnější paměti sousedních stránek v LAP bývá blízké („sousedi“)
  + princip lokality
  + zavádí se více stránek než se žádá
  + vhodné při inicializaci procesu
* Politika nahrazování - uplatňuje se v okamžiku, kdy je FAP plný
  + pokud není žádný rámec aktuálně označen jako volný, musíme nějaký uvolnit
  + je-li to nutné, před uvolněním stránky ji zapíšeme na disk
    - pokud se do ní nezapsalo, její kopie na disku je aktuální
    - zda se do stránky zapsalo zjistíme v bitu modify (dirty)
    - bit modify nastavuje HW automaticky při zápisu do stránky
  + určení oběti - politika přidělování místa
  + kolik rámců přidělit procesu?
  + intra/extra (local/global) množina možných obětí
    - oběti lze hledat jen mezi stránkami procesu, který vyvolal výpadek stránky?
    - oběti lze hledat i mezi stránkami ostatních procesů (např. podle priorit procesů)?
  + nelze obětovat kteroukoliv stránku
    - někde rámce jsou „zamčeny“ - typicky I/O buffery, řídicí struktury OS, …
* Stránkování na žádost
  + Stránka se zavádí do FAP jen když je potřebná
  + Výhody: méně I/O operací, menší požadavky na paměť, rychlejší reakce, může pracovat více uživatelů
  + Kdy je stránka potřebná?
    - byla odkazovaná
    - legální reference - nachází se ve FAP, přeloží se logická adresa na fyzickou
    - nelegální reference (porušení ochrany) → abort procesu
    - reference stránky, která není ve FAP → její zavedení do FAP - zprostředkovává OS

**BIT VALID-INVALID**

* S každým řádkem PT je spojen bit V/I (valid/invalid) (1 → je ve FAP, 0 → není právě ve FAP)
  + iniciálně jsou všechny V/I bity nastaveny na 0
* Při překladu adresy
  + jestliže je bit valid/invalid roven 0, generuje se přerušení typu „page fault“

**VÝPADEK STRÁNKY**

* Pokud se stránka nenachází ve FAP, je aktivován OS pomocí přerušení „page fault“ (výpadek stránky)
* OS zjistí:
  + nelegální reference -> proces je informován (např. signál SIGSEGV)
  + legální reference, ale stránka chybí v paměti → zavede ji
* Zavedení stránky
  + získání prázdného rámce
  + zavedení stránky do tohoto rámce
  + úprava tabulky, nastaven bit valid/invalid na 1
* Pak se opakuje instrukce, která způsobila „page fault“

**PRINCIP LOKALITY**

* Odkazy na instrukce programu a data mají tendenci tvořit shluky
* Časová lokalita a prostorová lokalita → lze dělat rozumné odhady o částech programu/dat potřebných v nejbližší budoucnosti
* vnitřní paměť se může zaplnit - něco umístit do FAP pak znamená nejdříve něco odložit z FAP

**ALGORITMY URČENÍ OBĚTI**

* Kritérium optimality -snaha o najmenšiu pravdepodobnosť výpadku stránky
* Algoritmus LRU (Least Recently Used) - oběť = nejdéle neodkazovaná stránka
  + princip lokality říká, že pravděpodobnost jejího brzkého použití je velmi malá (výkon blízký optimální strategii)
* Algoritmus FIFO (First-InFirst-Out) - oběť = stránka nejdéle zobrazená ve FAP
  + často používané stránky jsou v mnoha případech právě ty nejstarší
  + tj. algoritmus zbytečně odkládá často používané stránky
  + jednoduchá implementace - stačí udržovat ukazatelem vazbu do cyklického pořadí
* Algoritmus poslední/druhá šance - FIFO + vynechávání z výběru těch stránek, na které od posledního výběru bylo odkázáno
  + Odkazem na stránku stránka se nastaví příznak (use bit)
  + Oběť, která nemá nastaven příznak a je na řadě při kruhovém procházení je nahrazovaná

**I/O PORTY**

* HW pro I/O je značně rozmanitý, existují však určité běžně používané prvky
  + Port, sběrnice (bus), řadič (host adapter, controller)
* Adresy I/O zařízení
  + uváděné přímo v I/O instrukcích (např. IN AL, DX : DX port, AL získaný bajt)
  + I/O se mapuje na přístup k paměti (např. grafická karta, videopaměť)
* Základní způsoby ovládání I/O – polling(aktivní čekání na konec operace), přerušení, DMA
* I/O port se obvykle skládá ze 4 registrů
  + Data-in - Čtení vstupu od zařízení
  + Data-out - Pro zápis výstupu do zařízení
  + Status - Aktuální stav (data připravena, chyba, …)
  + Control - Ovládání zařízení, konfigurace, příkazy,…
* Programovaný I/O (busy-waiting, polling) - opakovaně se ptám na stav zařízení
  + připraven/ pracuje/ chyba
* I/O řízený přerušením
  + zahájení I/O pomocí I/O příkazu
  + paralelní běh I/O s během procesoru
  + I/O modul oznamuje přerušením konec přenosu
* Direct Memory Access (DMA)
  + kopírování bloků mezi pamětí a I/O zařízením na principu kradení cyklů paměti
  + přerušení po přenosu bloku (indikace konce)
  + nahrazuje programovaný I/O při velkých přesunech dat
  + vyžaduje speciální DMA řadič
  + při přenosu dat se obchází procesor, přístup do paměti zajišťuje přímo DMA řadič
  + procesor a DMA soutěží o přístup k paměti
* Jádro OS se snaží skrýt rozdíly mezi I/O zařízeními a programátorům poskytuje jednotné rozhraní
* Dále vrstva ovladačů ukrývá rozdílnost chování I/O řadičů i před některými částmi jádra
* Některé vlastnosti I/O zařízení
  + mód přenosu dat:
    - blokové zariadenia (disk)
      * příkazy: read, write, seek
      * možný přístup formou souboru mapovaného do paměti
    - znakové zariadenia (klávesnice, myš, sériový port)
      * příkazy: get, put
* sieťové zariadenia – prístup k nim sa značne líši, preto mávajú samostatné rozhrania OS
  + Unix I Windows obsahujú rozhrania “sockets” – prístup ako k súborom
  + způsob přístupu: sekvenční (modem) / přímý (disk)
  + sdílené/dedikované: klávesnice / páska
  + rychlost přenosu: vystavení, přenos, ...
  + read-write, read only, write only
* Blokující I/O
  + z hlediska procesu synchronní - proces čeká na ukončení I/O
  + snadné použití (programovaní), snadné porozumění
  + někdy však není dostačující (z důvodu efektivity)
* Neblokující I/O
  + řízení se procesu vrací co nejdříve po zadání požadavku
  + vhodné pro uživatelské rozhraní, bufferovaný I/O
  + bývá implementováno pomocí vláken
* Asynchronní I/O
  + proces běží souběžně s I/O
  + konec I/O je procesu hlášen signály
  + obtížné na programovaní, složité používání, ale v případě vhodně promyšleného programu efektivní

**I/O SUBSYSTÉM V JÁDRU**

* Plánování - některé I/O operace požadují řazení do front na zařízení
* Vyrovnání (vyrovnávací paměti), buffering **-** ukládání dat v paměti v době přenosu k/ze zařízení
  + řeší rozdílnost rychlosti avelikosti datových jednotek
* Caching - rychlá paměť udržuje kopii dat
  + caching je klíčem k dosažení vysokého výkonu
* Spooling - udržování fronty dat určených k výpisu na zařízení
  + pokud zařízení může vyřizovat požadavky pouze sekvenčně (typicky tiskárna)
* Rezervace zařízení **-** exkluzivita přístupu k zařízení pro proces
  + rezervace / uvolnění – volání systému
  + pozor na uváznutí (deadlock)
* CHYBOVÉ ŘÍZENÍ - Vzpamatování se po poruše při chybě čtení z disku, zjištění nedostupnosti zařízení, po náhodné chybě zápisu, …
  + Volání požadující I/O operaci získá číslo chyby
  + Udržuje se záznam o chybách v systému pro následné analýzy
* ZVYŠOVÁNÍ VÝKONU
  + Omezujeme počet přepnutí kontextu
  + Omezujeme zbytečné kopírování dat
  + Omezujeme počet přerušení tím, že přenášíme delší bloky
  + Využíváme všech výhod (funkcí) moderních řadičů
  + Používáme co nejvíce DMA
  + Všechny komponenty kombinujeme s cílem dosažení co nejvyšší propustnosti
    - CPU, paměť, sběrnice, I/O zařízení

**STRUKTURA DISKU**

* Diskové mechanismy se adresují jako velká 1-dimensionální pole logických bloků
* Doba přístupu (access time) je dána:
  + dobou vystavení (seek time) – na cylindr se stopou s adresovaným sektorem
  + dobou rotačního zpoždění – dodatečná doba do průchodu adresovaného sektoru pod čtecí/zápisovou hlavou
* Minimalizace doby vystavení
  + doba vystavení vystavovací vzdálenosti
  + řeší plánování činnosti disku
* Šířka pásma
  + počet přenesených bytů / doba od zadání skupiny požadavků do jejich ukončení

**PLÁNOVÁNÍ DISKU:**

* **FCFS –** opäť klasické FIFO – na ktorý cylinder narazí ako prvý, ten spracuje
* **SSTF** (shortest seek time first)
  + Z fronty požadavků vybírá ten, který vyžaduje minimální dobu vystavení od současné pozice hlavičky
  + algoritmus je variantou algoritmu SJF, může způsobit stárnutí požadavků.
* **SCAN**
  + Hlavička disku začíná na jedné straně disku a přesunuje se při splňování požadavků ke druhé straně disku. Pak se vrací zpět a opět plní požadavky. Někdy nazývané algoritmus typu výtah (elevator)
* **C-SCAN**
  + Hlavička se posouvá z jednoho konce disku na druhý a zpracovává požadavky. Potom se vrací zpět bez vyřizování požadavků a opět začíná vyřizovat požadavky z prvního konce.
  + Cylindry považuje za kruhový seznam, který za posledním cylindrem pokračuje opět prvním cylindrem. Poskytuje jednotnejšiu čakaciu dobu než SCAN.
* **C-LOOK**
  + Obdoba C-SCAN, ale hlavička jen potud do kraje, pokud existují požadavky, pak se vrací zpět
* SSTF je prirodzený, ale SCAN a C-SCAN jsou vhodnější pro těžkou zátěž disku – Výkon závisí požiadavkov
* Plánovací algoritmus by měl být napsán jako modul, aby plánovací algoritmus OS bylo možné zaměňovat

**PLÁNOVACIE ALGORITMY LINUX**

* Fronty v řadiči disku mohou měnit pořadí od OS
* Algoritmy
* Noop - Nemění pořadí požadavků
  + - Jen pokud nově příchozí navazuje na předchozí požadavek, požadavky budou sloučeny
  + Anticipatory
    - Jako deadline, ale přidává očekávání, že při sekvenčním přístupu k souboru přijde po jednom požadavku brzy požadavek následující. Proto chvíli vyčká
    - Neodlišuje požadavky čtení a zápisu
    - Odlišuje asynchronní požadavky od synchronních
  + Deadline
    - Vychází z algoritmu SCAN
    - Přidává deadline (do kdy má být požadavek vyřízen)
    - Po dávce požadavků se vzrůstajícími čísly sektorů (SCAN) kontroluje deadline
      * Pokud je třeba vytvoří speciální dávku pro vyřešení požadavků s expirovaným deadline.
    - Upřednostňuje čtení před zápisem
* CFQ (Completely Fair Queue) - Snaží se být spravedlivý k procesům
  + - Každý proces dostává určitý časový díl, kdy má exkluzivní přístup pro synchronní požadavky
    - Parametry
      * slice\_sync – délka slice v ms (bere v úvahu I/O prioritu procesu)
      * quantum – počet požadavků
    - 17 front (pro každou prioritu jedna) pro asynchronní požadavky
    - Každá fronta získává určitý slice a fronty jsou obsluhovány algoritmem RR

**TERCIÁLNE PAMETE**

* Dva aspekty
* šířka pásma (bandwidth) – měří se v B/s
  + - podporovaná šírka pásma – premierná rýchlosť prenosu dát behom veľkého prenosu
    - efektívna šírka pásma – priemer za celú dobu I/O operácie
  + zpoždění (latency) – doba potrebná pre vyhľadanie dát
* väčšina OS pracuje s vymeniteľnými diskami rovnako ako s pevnými diskami
* Pásky
  + jsou prezentované jako „holé“ (raw) paměťové médium
  + aplikace na páskách nemají k dispozici souborový systém
  + otevírají celý páskový mechanismus jako zařízení
  + páskové mechanismy se vesměs nesdílejí, jsou dedikované konkrétní aplikaci

**TECHNOLOGIE RAID**

* RAID: Redundant Arrays of Independent (Inexpensive) Disks
* organizace disků řízená tak, že poskytuje objem jednoho disku
  + s velkou kapacitou a rychlostí díky tomu, že mnoho disků pracuje paralelně
  + s velkou spolehlivostí, data se uchovávají redundantně, lze je obnovit i po poruše některého z disků
* Pravděpodobnost, že některý disk z množiny N disků selže je mnohem vyšší, než pravděpodobnost, že selže jediný disk
  + techniky na bázi redundance chránící před ztrátou dat jsou pro systémy s velkým počtem komponent (disků) kritické
* Původní záměr - levná alternativa nahrazující velké drahé disky
* Zrcadlení (stínování), Mirroring (shadowing) - každý disk je duplikován, 1 logický disk je tvořen 2 fyzickými
  + každý zápis se provede na obou discích, čte se z jednoho disku
  + jestliže se jeden disk porouchá, data jsou k dispozici na druhém disku
  + ke ztrátě dat dojde při výpadku obou disků, když zrcadlový disk selže dříve, než se systém opraví
* Dva hlavní cíle paralelismu v diskových systémech:
  + zvýšení propustnosti vyvážením zátěže malými přístupy
  + paralelizace velkých přístupů s cílem zkrácení doby odpovědi
* Zvýšení přenosové rychlosti paralelním zápisem do více disků (dělení, striping)
  + bit-level striping
    - dělení bitů každého bytu mezi samostatné disky
    - v poli 8 disků se zapisuje bit i každého bytu na disk i
    - čtení dat probíhá 8x rychleji než z jednoho disku
    - vystavení je delší než v případě jednoho disku
    - dnes se bit-level striping de facto už nepoužívá
  + blok-level striping
    - systém s n disky, blok souboru i se zapisuje na disk (i mod n) + 1
    - požadavky na různé bloky se mohou realizovat paralelně, pokud bloky leží na různých discích
    - požadavek na dlouhou posloupnost bloků může použít všechny disky paralelně
* RAID Level 0: Žádná redundance, jen souběžnost
* RAID Level 1: Spolehlivost dosažená zrcadlením disků
* RAID Level 2: Hamming code error correction
* RAID Level 3: 1 kontrolní disk na skupinu, dělení bitů
* RAID Level 4: Nezávislé operace read/write, dělení bloků
* RAID Level 5: Data/parity přes všechny disky (více souběžný přístup)
* RAID Level 6: Odolnost při více než jedné poruše disku

**Atomická operace** – taková operace, která vždy proběhne bez přerušení

* count++ - zdanlivo atomická operácia ale v assembleri je realizovaná až tromi inštrukciami

**RACE CONDITION** (podmínka soupeření)

* více procesů současně přistupuje ke sdíleným zdrojům a manipuluje s nimi
* konečnou hodnotu zdroje určuje poslední z procesů, který zdroj po manipulaci opustí
* Ochrana procesů před negativními dopady race condition ­- je potřeba procesy synchronizovat

**KRITICKÁ SEKCE**

* N procesů soupeří o právo používat jistá sdílená data
* V každém procesu se nachází segment kódu programu nazývaný kritická sekce, ve kterém proces přistupuje ke sdíleným zdrojům
* je potřeba zajistit, že v kritické sekci, sdružené s jistým zdrojem, se bude nacházet nejvýše jeden proces
* Podmínka vzájemného vyloučení (mutual exclusion), podmínka bezpečnosti, „safety“
  + jestliže proces P1 provádí svoji kritickou sekci, žádný jiný proces nemůže provádět svoji kritickou sekci sdruženou se stejným zdrojem
* Podmínka trvalosti postupu (progress), podmínka živosti, „liveliness“
  + jestliže žádný proces neprovádí svoji sekci sdruženou s jistým zdrojem a existuje alespoň jeden proces, který si přeje vstoupit do kritické sekce sdružené s tímto zdrojem, pak výběr procesu, který do takové kritické sekce vstoupí, se nesmí odkládat nekonečně dlouho
* Podmínka konečnosti doby čekání (bounded waiting), podmínka spravedlivosti, „fairness“
  + musí existovat horní mez počtu, kolikrát může být povolen vstup do kritické sekce sdružené s jistým zdrojem jiným procesům než procesu, který vydal žádost o vstup do kritické sekce sdružené s tímto zdrojem, po vydání takové žádosti a před tím, než je takový požadavek uspokojen
  + předpokládáme, že každý proces běží nenulovou rychlostí
  + o relativní rychlosti procesů nic nevíme

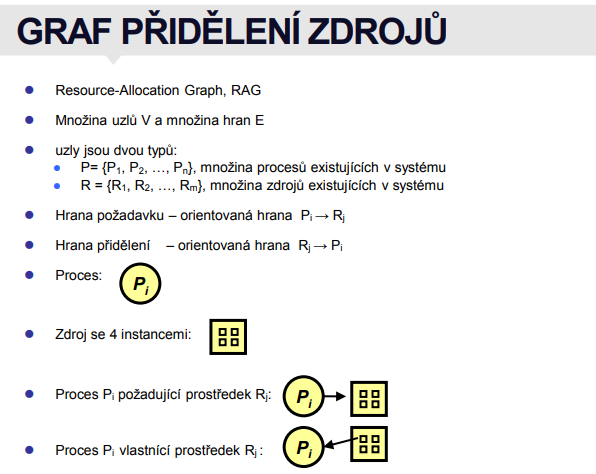
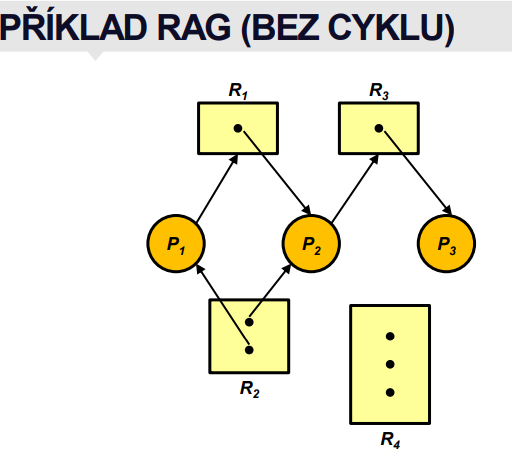
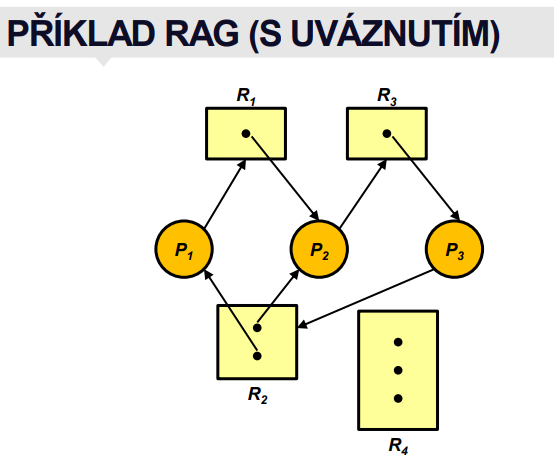
**ŘEŠENÍ PROBLÉMU KS**

* Softwarová řešení - algoritmy, jejichž správnost se nespoléhá na žádné další předpoklady
  + s aktivním čekáním „busy waiting“
* Hardwarová řešení - vyžadují speciální instrukce procesoru
  + s aktivním čekáním
* Řešení zprostředkované operačním systémem
  + potřebné funkce a datové struktury poskytuje OS
  + s pasivním čekáním
  + podpora v programovacím systému/jazyku - semafory, monitory, zasílání zpráv
* **SEMAFORY -** Synchronizační nástroj, který lze implementovat i bez „busy waiting“
  + proces je (operačním systémem) „uspán“ a „probuzen“
  + Má se provést akce B v Pj pouze po té, co se provede akce A v Pi
  + Použije se semafor flag inicializovaný na 0
  + Obecný semafor S **-** celočíselná hodnota z neomezovaného intervalu
  + Binární semafor - celočíselná hodnota z intervalu <0, 1> - ľahšia implementácia

**UVÁZNUTÍ**

* Uváznutí - dva nebo více procesů neomezeně dlouho čekají na událost, kterou může generovat pouze jeden z čekajících procesů
  + množina procesů P uvázla, jestliže každý proces Pi z P čeká na událost (uvolnění prostředku, zaslání zprávy), kterou vyvolá pouze některý z procesů P
* K uváznutí dojde, když začnou současně platit 4 následující podmínky
  + vzájemné vyloučení (mutual exclusion)
    - sdílený zdroj může v jednom okamžiku používat pouze jeden proces
  + ponechání si zdroje a čekání na další (hold and wait)
    - proces vlastnící zdroj čeká na získání dalšího zdroje, vlastněného jiným procesem
  + bez předbíhání (no preemption)
    - zdroj lze uvolnit pouze procesem, který ho vlastní
  + kruhové čekání (circular wait)
    - existuje takový seznam čekajících procesů (P0, P1, …, Pn), že P0 čeká na uvolnění zdroje drženého P1, P1 čeká na uvolnění zdroje drženého P2, …, Pn-1čeká na uvolnění zdroje drženého Pn, a Pn čeká na uvolnění zdroje drženého P0

**STÁRNUTÍ**

* Stárnutí - neomezené blokování, proces nemusí být odstraněný z fronty na semafor nikdy
  + požadavky 1 nebo více procesů z P nebudou splněny v konečném čase
    - z důvodů vyšších priorit jiného procesu
    - z důvodů prevence uváznutí apod
* Jestliže se v RAG nevyskytuje cyklus – k uváznutí nedošlo
* Jestliže se v RAG vyskytuje cyklus
  + existuje pouze jedna instance zdroje daného typu → k uváznutí došlo
  + existuje více instancí zdroje daného typu → k uváznutí může (ale nemusí) dojít

**PROBLÉM UVÁZNUTÍ**

* Ochrana před uváznutím prevencí
  + zajistíme, že se systém nikdy nedostane do stavu uváznutí
  + Nepřímé metody
    - zneplatnění některé nutné podmínky
      * Virtualizací prostředků, ruším nutnost vzájemné výlučnosti při přístupu
      * požadováním všech prostředků najednou
      * odebíráním prostředků
  + Přímé metody
    - nepřipuštění platnosti postačující podmínky (cyklus v grafu)
      * uspořádání pořadí vyžadování prostředků
  + Vzájemné vyloučení
    - podmínka není nutná pro sdílené zdroje
    - u nesdílených zdrojů musí podmínka platit
    - řeší se např. virtualizací prostředků (např. tiskárny)
  + Ponechání zdrojů a čekání na další
    - při žádosti o zdroje proces žádné zdroje „vlastnit“ nesmí
    - proces musí požádat o zdroje a obdržet je dříve než je spuštěn běh procesu
    - důsledkem je nízká efektivita využití zdrojů a možnost stárnutí
  + Zakázané předbíhání
    - jestliže proces držící nějaké zdroje a požadující přidělení dalšího zdroje, nemůže zdroje získat okamžitě, pak se uvolní všechny tímto procesem držené zdroje
    - „odebrané“ zdroje se zapíší do seznamu zdrojů, na které proces čeká
    - proces bude obnoven, pouze jakmile může získat jak jím původně držené zdroje, tak jím nově požadované zdroje
  + Zabránění kruhovému pořadí
    - zavedeme úplné uspořádání typů zdrojů a každý proces bude žádat o prostředky v pořadí daném vzrůstajícím pořadí výčtu
* Obcházení uváznutí
  + detekce potenciální možnosti vzniku uváznutí a nepřipuštění takového stavu
  + zamezujeme současné platnosti všech nutných podmínek
  + prostředek se nepřidělí, pokud by hrozilo uváznutí (hrozí stárnutí)
  + Algoritmus řešící obcházení uváznutí dynamicky zkouší, zda stav systému přidělování zdrojů zaručuje, že se procesy v žádném případě nedostanou do cyklické fronty čekání
* Obnova po uváznutí
  + uváznutí povolíme, ale jeho vznik detekujeme a řešíme
  + Udržuje se graf čekání (wait-for graph)
    - uzly jsou procesy
    - Pi → Pj jestliže Pi čeká na Pj
  + Periodicky se provádí algoritmus, který v grafu hledá cykly
  + Algoritmus pro detekci cyklu v grafu požaduje provedení n2 operací, kde n je počet uzlů v grafu
  + Násilné ukončení uváznutých procesů
  + Násilně se ukončuje jednotlivě proces po procesu, dokud se neodstraní cyklus
  + Výběr oběti: minimalizace ceny
  + Návrat zpět (rollback) – návrat do některého bezpečného stavu, proces restartujeme z tohoto stavu
  + Stárnutí – některý proces může být vybírán jako oběť trvale
* Ignorování hrozby uváznutí
  + uváznutí je věc aplikace ne systému
  + způsob řešení zvolený většinou OS