

OS je program, ktorý funguje ako spojka medzi užívateľom počítača a hardware. Spravuje a riadi využívanie všetkých zdrojov počítača (prideľuje ich, rozhoduje medzi konfliktnými požiadavkami).

Ciele OS:

- Efektívne využívať hardware počítača
- Riadiť užívateľské (aplikačné) programy – ktoré pracujú na vysokej úrovni a využívajú len **volanie služieb OS** (nič iné nevidia - nemôžu pristupovať k HW priamo)
- Učiniť počítač jednoduchšie a bezpečnejšie použiteľný

Klasifikácia počítačov:

- I. Mainframe (strediskový počítač) – radil úlohy do **dávok** (batch) = prebehol rýchly výpočet, ale výstup na tlačiareň trval dlho – CPU zbytočne čakal => prvé koncepty **multiprogramového režimu** riešenia viacerých úloh – potrebné implementovať:
 - o ovládanie I/O: prerušenie; privilegované a nepriviligované inštrukcie
 - o správa pamäti: dynamické prideľovanie pamäti daným procesom; fyzický a logický adresový priestor (FAP a LAP); ochrana pred neautorizovaným prístupom
 - o plánovanie CPU: intervalový časovač (po uplynutí času generuje prerušenie) – OS musí byť schopný zvoliť z pripravených procesov
 - o prideľovanie zdrojov
- II. Osobný počítač – pre jedného užívateľa – kompletne I/O vybavenie, multitasking
- III. **Paralelné** (viac procesorov, spoločný FAP = úzko previazané) a **distribúované** systémy (viac počítačov, komunikujú formou predávania správ = voľne previazané => sieťové infraštruktúry LAN, WAN => model P2P alebo K/S)
 - o **SMP** (symmetric multiprocessing) = viac procesov beží naraz na rôznych procesoroch; podporuje väčšina moderných OS
 - o **AMP** (asymmetric multiprocessing) = každý procesor má svoju úlohu (master/slaves)

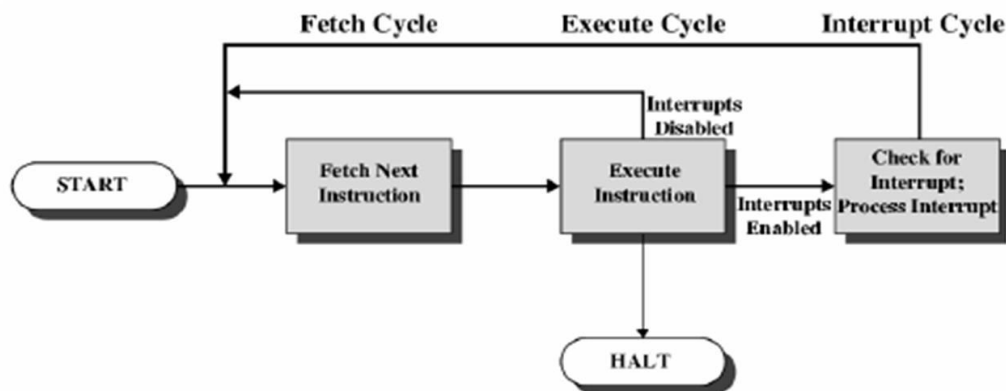
Flynnova kategorizácia výpočtových systémov:

- SISD = Single Instruction Single Data – klasický 1-procesorový systém, OS s multitaskingom
- SIMD = Single Instruction Multiple Data – vektorové procesory
- MISD = Multiple Instruction Single Data – každý procesor vykonáva nad rovnakými dátami iné operácie; nikdy prakticky neimplementované
- MIMD = Multiple Instruction Multiple Data – tesne a voľne previazané systémy

Procesor: získava inštrukcie z operačnej pamäte, dekoduje ich a sekvenčne vykonáva

- **Inštrukcie:** operácie nad hodnotami, skoky, riadenie I/O, zápis/čítanie z pamäti
- Množina inštrukcií je špecifická pre každý typ procesoru (CISC/RISC)
- Je vybavený internými pamäťami s malou kapacitou a rýchlou dobou prístupu: **registre** –
 - delia sa na **viditeľné užívateľovi** (obsahujú dáta, adresy, flagy, ukazatele stacku...)
 - a **stavové a riadiace** – obecné nedostupné užívateľským procesom (*PC* pre riadenie vykonávania programu – ukazateľ na nasledujúcu inštrukciu, *IR* pre riadenie vlastných operácií CPU)
- Medzi procesorom a operačnou pamäťou: cache pre rýchle sprístupňovanie dát / inštrukcií
- Zbernica: komunikačný prostriedok (je n-bitovo paralelný: 8, 16, 32, 64 komunikačných ciest) medzi CPU, pamäťou a perifériami pre prenos adres a dát.

Instruction Cycle with Interrupts



TSS = Time Sharing Systems (multitasking): periférne zariadenia aj CPU môžu pracovať súbežne.

- Radič perfiérie: zodpovedá za jej chod, riadi ju pomocou signálov
- Každý radič periférií ma svoju vyrovnávaciu pamäť – *buffer*
- CPU (alebo DMA, viz. neskôr) presúva dáta z bufferu do operačnej pamäte a späť
- Radič perfiérie informuje CPU o svojej činnosti **PRERUŠENÍM** alebo zmenou v registroch.

Prerušenie behu procesu – dynamicky vzniká potreba vykonať inú postupnosť príkazov ako reakcia na nejakú udalosť. Potlačí sa vykonávanie bežiaceho procesu tak, aby ho neskôr bolo možné obnoviť.

- CPU predá riadenie správcovi prerušení - Interrupt Handler Routine prostredníctvom vektoru prerušení – ten obsahuje adresy vstupných bodov všetkých správcov prerušení
- Počas prerušenia sú ďalšie prerušenia maskované
- Zapamätá sa stav CPU - uchováva sa čítač inštrukcií a registry
- Testuje nutnosť prerušenia po každej inštrukcii

Metódy obsluhy I/O:

- **Busy waiting:** v systémoch bez riadenia I/O pomocou OS
 - žiadne súbežné spracovávanie požiadaviek (blokujúci I/O)
 - nevyriešená zostáva najviac jedna požiadavka
 - program testuje koniec I/O operácie opakovanými dotazmi (**polling**) na príslušný stavový register I/O zariadenia
- **Prerušením:** v systémoch s riadením I/O – OS zahajuje I/O operáciu na žiadosť procesu:
 - *synchronne* (proces čaká na dokončenie I/O) alebo
 - *asynchronne* (proces nečaká na dokončenie I/O – ak napr. vstup príliš dlho trvá)

DMA sa používa pre veľmi rýchle I/O zariadenia, keď CPU nestíha dobehnúť rýchlosť prenosu dát (sieťové / grafické karty), alebo keď CPU vykonáva inú prácu a nemôže sa zaťažovať I/O operáciami. CPU iniciuje prenos dát (odkiaľ a koľko), DMA radič perfiérie potom prenáša bloky dát medzi RAM a perifériou **priamo**, bez ďalších zásahov CPU => **cycle stealing** (kradnutie inštrukčných cyklov CPU pre komunikáciu s pamäťou). Prerušenie: po prenesení celého bloku dát (nie napr. po prenesení 1B).

Štruktúra pamäte: primárna - operačná = adresovateľná matica buniek, CPU do nej zasahuje priamo, volatilná, rýchlosť prístupu v desiatkach nanosekúnd. Sekundárna – magnetický disk; nonvolatilná, rýchlosť prístupu v milisekundách. Ternárna – archívne médiá – max. kapacita, najpomalší prístup: napr. WORM (Write Once, Read Many times).

Bezpečnostné mechanizmy v OS: nesprávny program nesmie negatívne ovplyvniť iné programy

- 2 režimy procesoru: **user mode** – obmedzený inštrukčný repertoár, aby užívateľský program nesiahahal na pamäť a zdroje, ktoré nie sú jeho
a kernel mode – neobmedzený + privilegované inštrukcie – aktivuje sa po prijatí prerušenia – naspäť do user mode zase špecifickou privilegovanou inštrukciou
 - užívateľský proces nikdy nemôže získať riadenie v privilegovanom režime
- **Ochrana I/O** – všetky I/O inštrukcie sú privilegované
- **Ochrana pamäte** pred zápisom na nepovolené miesto (viz. správa pamäte)
- **Ochrana dostupnosti CPU** – časovač (viz. plánovanie)

Typológia OS:

- MAINFRAME OS = pre dátové centrá, databáze => množstvo periférií (tisíce diskov, terabajty dát), množstvo súbežne bežiacich procesov; orientácia na Linux
- EMBEDDED OS = v autách, mikrovlnkách, DVD prehrávačoch... užívateľ nemá možnosť nič inštalovať, všetko je v ROM + orientácia na real-time činnosť; QNX, VxWorks
- MULTIPROCESSOR OS = súbežnosť viacerých funkcií OS, pre počítače s viacjadrovým CPU
- PERSONAL COMPUTER OS = podpora multitaskingu, jeden užívateľ (Win 7, Mac OS, Linux)
- HANDHELD OS = pre Smartphone, PDA: nemajú vonkajšiu pamäť, bežne spúšťajú aplikácie tretích strán (nie vždy dôveryhodné), podpora komunikácie a multitaskingu
- SERVER OS = veľký výkon, pamäťová aj komunikačná kapacita – poskytovanie služieb množstvu klientom; Solaris, Linux, FreeBSD, Windows Server
- SENSOR NODE OS = pre uzly senzorových sietí, malá pamäť, nutná veľká výdrž batérie, bezdrôtová komunikácia, všetko je nainštalované vopred; TinyOS
- REAL-TIME OS = jednoúčelové, riadenie dedikovaných aplikácií (vedecké prístroje, riadenie priemyselných procesov, monitorovacie systémy...) – napr. **SOC** (System on a Chip) - nemajú interpret príkazov
 - **Striktné**, Safety-Critical OS – často obmedzená/žiadna vonkajšia pamäť, definované časové limity pre spracovanie inštrukcií, ktoré musia byť deterministicky dodržané
 - **Tolerantné**, Soft-RT OS – nemusí vždy splniť časový limit – povoľuje odchýlky, prioritný proces má vždy prednosť: preemptívne plánovanie

Generické komponenty OS:

- Správa **procesora**, procesov a vlákien - ich krátkodobé plánovanie **dispečerom**:
 - vytváranie, rušenie, pozastavovanie a obnova behu procesov a vlákien (multitasking)
 - ich synchronizácia a komunikácia medzi nimi, zvládnutie uviaznutia
 - *jednovláknové* procesy: jediný čítač inštrukcií, sekvenčné vykonávanie inštrukcií
 - *viacvláknové*: jeden čítač inštrukcií pre každé vlákno, súbežné vykonávanie vlákien, ktorých inštrukcie sú vykonávané sekvenčne
- Správa operačnej **pamäte** = jej prideľovanie a uvoľňovanie na žiadosť, virtualizácia
 - zisťovanie, ktorý proces v danom okamihu používa ktorú časť FAP

- FAP = adresový priestor operačnej pamäte, pole samostatne adresovateľných jednotiek, repozitár inštrukcií a dát
- LAP = to, čo na logickej úrovni patrí procesu; transformácia adres LAP na FAP (t.j. presun dát do fyzickej pamäti) až pri vykonávaní danej inštrukcie v CPU
 - Lineárna štruktúra: pole **stránok**, virtualizácia stránkovaním
 - alebo 2D pole lineárnych **segmentov**, virtualizácia segmentovaním
- Lineárny LAP môže byť zobrazený do FAP identitou, alebo pomocou HW:
- **DAT** (Dynamic Address Translation) a **MMU** (Memory Management Unit): pri odkázaní na miesto s adresou LAP, ktoré nie je vo FAP hľadá (alebo vytvorí) vo FAP voľný blok, kde prenesie požadované dáta
- Správa **I/O** = ovládače, cache pamäte, *pooling* (prekrývanie výstupov 1 procesu a vstupov 2.)
- Správa **súborov** a vnútornej (diskovej) pamäte = file system
 - logické jednotky – **súbory** - kolekcia súvisiacich informácií, člení sa na záznamy
 - vytváranie, úprava, mazanie, skladovanie, archivácia, obnova súborov a adresárov
 - spolupráca so sekundárnou pamäťou: správa a plánovanie diskových operácií
- Systém **ochrán** = riadenie prístupu k zdrojom, rozlišovanie ne/autorizovaných prístupov (t.j. užívateľské práva: uid, gid), ochranné opatrenia proti vnútorným a vonkajším útokom
- **Sieťovanie**, distribuované systémy = kooperácia vzdialených procesorov, zdieľanie zdrojov
- **Interpret užívateľských príkazov** na služby OS (command-line interpreter, shell)
- **Systémové programy** = stavové informácie, podpora jazyka a komunikácie, manipulácia so súbormi, databáze, ...

Rozhranie OS:

- Pre užívateľa: **CLI** (Command Line Interface, shell) / **GUI** (sú „user friendly“: okná, ikony...)
- Pre procesy: volajú služby OS (napr. knižnice jazyka C), alebo namiesto priameho volania s OS komunikujú cez **API** (napr. Win32 API, POSIX API, Java API pre Java Virtual Machine) – OS spracuje požiadavku a vráti výsledok bez toho, že by užívateľský program videl, čo OS robí
- **DLL** = podprogramy z knižníc, ktoré sa zavádzajú do programu počas jeho behu (= dynamicky)

Služby OS:

Riadenie procesov: zavedenie programu do pamäte a štart jeho riešenia + ukončenie (štd./s chybou)

- **fork()** = generovanie nového procesu - potomka - identického s rodičom
- **exec()** = štart procesu, **wait()** = počkaj, **end()**, **abort()** = ukonči sa štandardne/násilne
- detekcia chýb + chybové hlásenia
- **IPC** (Interprocess communication) – výmena informácií medzi procesmi:
 - v rámci 1 počítača alebo viacerými na sieti (zdieľaná pamäť / predávanie správ)
 - informácie sú posielané do a vyberané zo schránky – **socket** (v UNIXe: pipe)
- Predávanie parametrov medzi bežiacim procesom a OS:
 - Cez registre – sú dostupné procesu aj OS
 - Tabuľkou v operačnej pamäti – adresa tabuľky sa umiestni do registru
 - Cez zásobník – tiež je dostupný procesu aj OS (program – push, OS – pop)

Funkcie umožňujúce **manipuláciu so súbormi**: create, delete, open, close, read, write, mkdir, rmdir.

Funkcie pre **správu I/O zariadení**: drivery – request/release device, select, read, write.

Údržba informácií: get, set time/date/system data

Vnútorne služby OS: (nepomáhajú priamo užívateľovi, slúžia pre efektívnu prevádzku systému)

- **Resource Allocation** – prideľovanie prostriedkov / zdrojov
- **Accounting** – prehľad o tom, koľko ktorých zdrojov systému daný užívateľ používa
- **Protection** – ochrana a bezpečnosť

ARCHITEKTÚRY OS:

1. Vrstvený: dekompozícia – každá úroveň rieši konzistentnú množinu funkcií

- entity v jednej vrstve komunikujú pomocou *protokolov*
- nižšia vrstva ponúka vyššej vrstve základné služby
- nižšia vrstva nemôže vyžadovať vykonanie služby od vyššej vrstvy
- + vrstvu je možné modifikovať bez toho, že by to ovplyvnilo iné, jednoduchosť ladenia
- - obtiažnosť definície hraníc – tzv. **rozhraní** vrstiev, vznikajú kolízie

2. Idea kernelu: UNIX = systémové programy (shell, kompilátory) + jadro (obstaráva plnenie služieb z oblastí správy CPU, operačnej pamäte a I/O)

- **Monolitické jadro:** entita v privilegovanom režime so zamaskovaným prerušením
 - využíva špeciálne rysy HW nedostupné procesom
 - mohutné, previazané, ťažko manipulovateľné – pretože obsahuje všetky služby OS
 - pre každú volanú službu existuje jedna procedúra, ktorá ju rieši
 - správa sa ako server, ktorý postupne (po jednej) vybavuje požiadavky procesov
 - všetko mimo jadra je riešené formou **procesov** (systémové a užívateľské)
 - komunikácia medzi procesmi formou **predávania správ**
 - napr. UNIX, MS-DOS, Windows XP
- **Mikrojadro:** minimalistická varianta jadra - Microkernel System Structure:
 - OS je **kolekciou systémových procesov**, ktoré poskytujú jednotlivé služby
 - V mikrojadre ostáva len práca s pamäťou, procesmi a vláknami, IPC a I/O
 - Minimum funkcií v privilegovanom režime – ich presun do užívateľskej oblasti (drivery, služby súborového systému, virtualizácia pamäte)
 - Potom krach v drivere spôsobí pád jedného procesu a nie celého systému
 - Jadro sa správa len ako „ústredňa“ - systémové procesy separuje a prepojuje
 - Mikrojadro prináša vyššiu **spoľahlivosť, bezpečnosť** (menej programov v jadre), **rozšíriteľnosť** (jednoduché pridávanie a odstraňovanie služieb) a **prenositeľnosť** (na nové architektúry procesorov). Podporuje distribuovanosť (multiprocesory) a OOP.
 - Za cenu vyššej réžie – volanie služieb nahrádza výmena správ medzi procesmi
- **Bootstrapping:** spustenie činnosti počítača zavedením jadra do operačnej pamäte
- **Bootstrap loader** je uchovávaný v EPROM na matičnej doske (napr. BIOS v PC) – vie nájsť obraz jadra na vonkajšej pamäti
- Inicializácia jadra: zistenie skutočnej veľkosti operačnej pamäte, konfigurácia HW (grafická karta, myš, disky, ...), inicializácia virtuálnej pamäte, nastavenie vektoru prerušení, povolenie prerušenia, vytvorenie a spustenie počiatočného procesu (*init* v UNIXe, *smss* vo Win)

3. Modulárna architektúra: vytvorenie jadra pomocou OOP. Komponenty jadra sú jednotlivé moduly, ktoré medzi sebou komunikujú cez *rozhrania*. Každý modul sa *zavádza*, keď je potrebné ho použiť. Napr. mikrojadro **Mach** – zodpovedá za RPC, IPC, správu pamäte, dispečer + jadro **BSD**.

4. Virtualizácia: metóda, ktorá nám umožňuje: pozeráť sa na fyzickú entitu ako na viac logických entít alebo zlučovať viac fyzických entít do jedného virtuálneho celku.

- **Systém:** kompletne výpočtové prostredie zložené zo zdrojov (výpočtových, pamäťových a vstupno-výstupných) a operácií, ktoré nad nimi môžeme vykonávať
- **Virtuálny stroj:** na jednom *fyzickom* hostujúcom systéme súbežne beží viac *virtuálnych* systémov – do nich môžeme inštalovať bežné (multitaskingové) OS a spúšťať aplikácie
 - V podstate multitasking na úrovni výpočtových prostredí
 - Hostovaný OS neberie ohľad na iné hostované OS (pretože je od nich izolovaný) a predpokladá, že má exkluzívny prístup k zdrojom
 - Nutné riadenie všetkých hostovaných systémov manažérom – **hypervízorom**:
 - Musí umožniť vytvorenie/zrušenie hostovaného systému
 - Zaisťuje plánovanie a pridelovanie zdrojov, sprostredkovanie privilegovaných služieb hostovaným systémom a izoláciu jednotlivých systémov
 - Hostované systémy medzi sebou môžu komunikovať len po sieti ako vzdialené entity
 - Ak dôjde k poškodeniu alebo infiltrácii jedného systému, neovplyvní to ostatné
 - Hypervízor môže bežať „na hardware“ alebo ako aplikácia vo virtuálnom stroji
- Implementácia:
 - Virtuálny stroj beží vo *fyzickom užívateľskom* režime hostujúceho počítača
 - Musí sa ale správať ako samostatný systém – má mať *virtuálny užívateľský* aj *virtuálny privilegovaný* režim
 - Proces bežiaci vo virtuálnom užívateľskom režime zavolá službu OS
 - Prerušenie aktivuje virtualizačný software bežiaci v privilegovanom režime hostiteľa
 - Tento software zmení registre príslušného virtuálneho stroja, reštartuje stroj a zdelí mu, že beží vo virtuálnom privilegovanom režime
- Napr. **VMWare, Xen**
- **.NET** (Microsoft) = virtuálny stroj, pre ktorý je možné písať programy nezávisle na OS počítača, ktorý tento virtuálny stroj hostuje. Programy napísané v C# (a C++, F#) sa kompilujú na tzv. assemblies, ktoré sú just-in-time preložené do natívneho kódu hostujúceho systému.
- **JVM** (Java Virtual Machine) implementovaný pre každú platformu. Kompilátor generuje tzv. bytecode (typu *.class), nie strojový kód konkrétnej architektúry. Bytecode beží omnoho pomalšie ako ekvivalentný program v C – riešenie: just-in-time kompilátor.

6. Architektúry klient-server s viacerými vláknami

- **Thread per request** – pre riešenie každej novej požiadavky klienta sa vytvorí nové vlákno – po splnení služby sa samo zruší – vyššia priepustnosť, ale aj réžia
 - **Thread per connection** – server vytvorí jedno vlákno pre každé spojenie s jedným klientom a jeho požiadavky rieši sekvenčne – po uzavrení spojenia sa vlákno zruší
 - **Thread per object** – samostatné vlákno na každý sprístupňovaný objekt serveru
-

PROGRAM – súbor obsahujúci inštrukcie a dáta

- Sekvenčný model: program = 1 proces, alebo paralelný model: program > 1 proces

PROCES – jednotka realizácie výpočtu podľa programu, resp. abstrakcia prechodu programom

- charakterizovaný svojim kontextom (program counter) a adresovým priestorom (FAP), ktorý je mu pridelovaný

- môže vlastniť zdroje: I/O zariadenia, súbory, alebo komunikačné kanály k iným procesom
- prideľuje sa mu čas procesoru
- pozostáva z jedného alebo viacerých **vlákien** (= abstrakcia sekvenčnej činnosti procesu)
- **stavy**: new (iniciálny stav) / running (beží na CPU) / waiting / ready / terminated
- stavové informácie procesu: registre + čítač inštrukcií
- + informácie nutné pre správu procesov: PID (process identifier), jeho priorita, stav procesu, informácie o používaných zdrojoch
- = **PCB** (Process Control Block), resp. TCB (Task Control Block)
- Klasické (heavy-weight) procesy – všetky dáta sú privátne, zdieľaný je len program
- Ľahké (light-weight) procesy, vlákna (threads) – minimum vlastných dát, väčšina zdieľaných

LAP procesu: **halda** (rastie nahor – k vyšším adresám, má dynamický obsah)

+ **zásobník** (rastie nadol; typicky 1 pre každé vlákno – LIFO: pre ukladanie návratovej adresy pri volaní funkcie, ukladanie lokálnych premenných...)

- neprekrývajúce sa **oblasti** dostupné vláknam procesu
- každá oblasť má svoj priestor vymedzený báзовou adresou a dĺžkou + prístupové práva rwx pre jednotlivé vlákna
- stránkovaním je zobrazená do FAP
- veľkosť sa môže v dobe behu procesu meniť
- + pomocné **Shared Memory Regions** (zdieľanie pamäti medzi procesmi)

Vytvorenie procesu:

- pri bootovaní – init 1
- rodič vytvára ďalšie procesy – potomkov dvoma spôsobmi:
 - „na zelenej lúke“ – do pamäti sa zavedie text programu a dáta, vytvorí sa prázdna halda a inicializuje sa PCB (napr. vo Win API) – proces sa sprístupní dispečerovi
 - klon existujúceho procesu – presné kopírovanie dát a PCB (**fork** v UNIX-e) – procesy sa dozvedia, ktorý z nich má úlohu rodiča a ktorý je potomok
- po vykonaní fork sa často používa **exec** = zahod' pamäťový priestor zduplikovaného procesu a na jeho mieste spusti iný proces
- to je niekedy neefektívne, preto sa využíva princíp **Copy on write** – iníciaľne nový proces zdieľa stránky s pôvodným, až keď chceme zapisovať do stránky, vytvorí sa jej kópia
- rodič a potomok môžu zdieľať všetky zdroje, časť z nich alebo fungovať úplne samostatne
- rodič a potomok môžu bežať súbežne, rodič čaká na dokončenie potomka (wait) alebo potomok čaká na skončenie rodiča (a stáva sa z neho zombie)

Ukončenie procesu: po vykonaní poslednej inštrukcie požiada OS o svoje ukončenie volaním služby **exit** (ak zavolá exit jedno z vlákien procesu, končí celý proces!) a uvoľňujú sa zdroje. O ukončenie procesu môže požiadať jeho rodič = **abort**, ak napr. potomok prekročil isté množstvo povolených zdrojov, alebo rodič končí svoju existenciu. Proces môže skončiť aj *chybne*.

Prepínanie procesu (**context switching**) – základ multitaskingu = manipulácia s procesom:

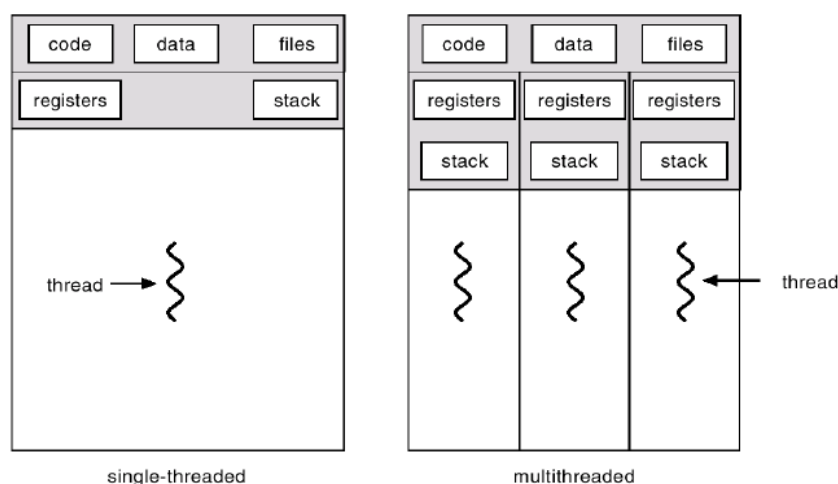
- prišlo prerušenie, procesu odoberáme procesor
- uchováme si jeho stav v **zozname PCB** – tabuľka pre každý proces
- a vyberieme iný proces z *fronty pripravených procesov* - potrebné **PLÁNOVAČE**:

- krátkodobý (**dispečer**): v situácii, keď sa uvoľní procesor, musí sa **rýchlo** rozhodnúť, kto z pripravených (ready) nastúpi – spúšťa sa najčastejšie (implementácia: červenočierny strom)
 - ak bežiaci proces prechádza do stavu čakajúci alebo končí, volá sa príslušná služba OS (synchronne, **nepreemptívne** plánovanie – postupuje sa podľa plánu)
 - ak bežiaci alebo čakajúci proces prechádza do stavu pripravený, iniciuje sa prerušenie (asynchronne, **preemptívne** plánovanie – s predbiehaním)
- strednedobý (**taktický**): zodpovedá za **swapping** + zabraňuje thrashingu (viz. virtuálna pamäť)
 - vyberá, ktorý proces môže byť v hlavnej pamäti / je treba zaradiť medzi *odsunuté* (lebo príliš mnoho procesov vo FAP znižuje výkon systému)
 - odoberá mu priestor vo FAP, dáva ho na disk: **swap-out**
 - proces je „odložený čakajúci“
 - resp. ktorému „odloženému pripravenému“ procesu vráti FAP: **swap-in**
 - proces prechádza do stavu pripravený
- dlhodobý – strategický (**job scheduler**): vyberá, ktorá požiadavka na výpočet bude zaradená medzi procesy a súťažiť o procesor, je vyvolávaný riedko

Vlákno = sekvenčný dej definovaný v procese

- viditeľné len v rámci konkrétneho procesu
- môže pristupovať k zdrojom, ktoré vlastní tento proces
- vlákno sa tiež nachádza v stavoch (running/ready/waiting), jeho kontext je uložený v TCB
- všetky **vlákna v procese sa riešia súbežne** – máme napr. textový editor: ak by bol riešený len sekvenčne, tak počas ukladania dát by ignoroval vstupy z klávesnice
- umožňuje efektívne využitie viacerých jadier procesora (vláknité programovanie)
- pre OS jednotka plánovania, nie zdrojov – skupina vlákien jedného procesu zdieľa premenné
- nemôžeme odkladať jedno vlákno – swapuje sa celý proces
- výhody:
 - paralelizmus, štrukturalizácia programu
 - rýchlejšie vytvorenie, ukončenie aj prepnutie kontextu ako pri procese
 - zdieľanie zdrojov (ale pozor na komunikáciu medzi vláknami)
 - interakcia (napr. požiadavky na server = vytvárajú sa vlákna)

Jednovláknové a vícevláknové procesy



- **ULT** (User-level Threads) – model N:1
 - správa vlákien na aplikačnej úrovni pomocou knižnice (**thread library**)
 - obsahuje funkcie pre vytváranie/rušenie vlákien, predávanie správ a dát medzi nimi, plánovanie ich behu a uchovávanie a obnovu kontextu
 - výhody: jadro o ich existencii nevie => pri prepínaní vlákien nie je nutné prerušovať, prepínať režim procesoru a volať služby jadra
 - aplikácia si sama volí najvhodnejší algoritmus
 - nevýhody: pri volaní služieb sa blokuje celý proces - sú nedostupné všetky vlákna
- **KLT** (Kernel-level Threads) – model 1:1
 - komplet celú správu vlákien podporuje API v jadre – nepoužíva sa knižnica
 - výhody: jadro môže súčasne plánovať beh viacerých vlákien jedného procesu na viacerých procesoroch + k blokovaniu dochádza na úrovni vlákien
 - prakticky sa používa v **kombinácii** s ULT – model N:M
- skupina vlákien pripravených na použitie = **bank** (thread pool)

PLÁNOVANIE PROCESOV

Plánovacie rozhodnutie vzniká, keď proces:

- vzniká a radí sa medzi **ready**
- prechádza zo stavu **run** do **waiting**
- prechádza zo stavu **waiting** do **ready**
- končí

Požiadavky na plánovanie:

- **časová proporcionalita**: rozumne vyberať procesy, ktoré budú bežať a ktoré budú prerušené (mail odoslaný o 3 sekundy po potvrdení – OK, korekcia obrazovky o 3 sekundy neskôr už nie)
- rovnomerné využívanie zdrojov – udržovanie systému v činnosti
- **spravodlivosť** – porovnateľné procesy musia získať porovnateľnú časť procesoru
- vysoká **priepustnosť** (počet procesov, ktoré dokončia svoj beh za jednotku času) vs. minimalizácia **doby čakania** (obrátky) procesu v stave ready
- rýchla **interakcia**

Váha kritérií podľa konkrétneho systému:

- Interaktívne: minimalizácia doby čakania, rýchla interakcia, proporcionalita
 - Algoritmy **RR** (Round Robin), **FSS** (Fair Share Scheduler), + prioritné plánovanie
- Dávkové systémy: maximálna priepustnosť, maximalizácia využitia CPU
 - Algoritmy **FCFS**, **SJF** (Shortest Job First), **SRTF** (Shortest Remaining Time First)

1.) First-Come – First-Served:

- Nepreemptívny, jednoduchá implementácia
- Ak sa ako prvý dostane veľký proces a veľa malých za ním čaká: **konvojový efekt**

2.) Shortest Job First:

- K definícii procesu sa doplní dĺžka jeho nasledujúcej CPU dávky

- Nepreemptívna varianta: ak sa proces dostane na rad, nemôže byť predbehnutý žiadnym iným procesom, pokiaľ svoju dávku CPU nedokončí
- Preemptívna varianta: ak sa vo fronte procesov objaví ready proces s dĺžkou dávky CPU kratšou ako je doba zostávajúca k dokončeniu dávky práve bežiacieho procesu, nový proces predbehne práve bežiaci proces – táto varianta sa nazýva **SRTF**
 - o optimálny, ak je pri plánovaní rozhodujúca minimálna priemerná doba čakania
- Bežiaci proces ale nemusí vždy využiť celé svoje pridelené kvantum času (napr. vyvolá I/O operáciu a skončí skôr)
- Dĺžku jeho nasledujúcej dávky je možné odhadnúť z **histórie chovania procesu**
 - o Musia byť zaznamenané predošlé odhady dĺžok dávok CPU
 - o A tiež ako ich proces v skutočnosti využil
 - o Heuristika - použije sa **exponenciálne priemerovanie**:
 - t_n ... skutočná dĺžka n-tej dávky CPU
 - T_{n+1} ... odhad dĺžky nasledujúcej n+1. dávky
 - α ... vplyv histórie, α je z intervalu $[0, 1]$
 - čím je menšie, tým má história na odhad menší vplyv
 - čím je väčšie, tým viac sa rešpektuje krátka história
 - $T_{n+1} = \alpha * t_n + (1 - \alpha) * T_n$
 - Formulu môžeme rozvinúť pomocou sumy až do T_0 , kt. sa zvolí ako konštanta
 - Keďže $1 - \alpha, \alpha \leq 1$, každý term má na T_{n+1} menší vplyv ako jeho predchodca

3.) Round Robin – cyklické plánovanie:

- Preemptívne plánovanie typu FCFS založené na sledovaní časových intervalov
- Každý proces dostáva CPU *cyklicky* na malú jednotku času (cca 100 ms) – **časové kvantum q**
- Po uplynutí doby q:
 - o je bežiaci proces predbehnutý najstarším procesom vo fronte
 - o sám sa zaradí na koniec tejto fronty
- Ak je vo fronte n procesov, každý získava $1/n$ -tinu doby (výkonu) CPU
- Žiadny proces nečaká dlhšie ako $(n-1) * q$ časových jednotiek
 - o Ak je q veľmi veľké, plánovanie sa blíži princípu FCFS
 - o Ak je q veľmi malé, CPU sa venuje prevažne prepínaniu kontextu procesov
 - Nech doba prepnutia kontextu je rovná 0.01
 - Režijné straty pri $q = 12, 6$ a 1 sú rovné 0.08%, 0.16% a 1%

4.) Prioritné plánovanie:

- S každým procesom je spojené prioritné číslo – integer
- CPU sa prideľuje procesu s najvyššou prioritou (typicky najnižšie číslo)
- Preemptívna varianta: keď sa vo fronte objaví proces s vyššou prioritou, než je priorita práve bežiacieho procesu – predbehne ho
- Procesy s veľmi malou prioritou sa ale **nikdy nemusia dostať na rad – problém starnutia**
- Riešenie: **zrenie procesov** – ich priorita sa po čase zvyšuje
- Napr. Linux:
 - o každému procesu prideli istý počet kreditov
 - o pri každom prerušení proces stráca 1 kredit – proces s 0 kreditmi sa vzdáva CPU
 - o ak neexistuje žiadny ready proces s kreditmi, všetkým procesom sa nastaví kredity na hodnotu: $\text{kredity}/2 + \text{priorita}$

- + real time algoritmus pre úlohy, kde je dôležitejšia ich priorita pred spravodlivosťou

5.) Plánovanie s viacúrovňovými frontami:

- **Prednostná** fronta (foreground) – interaktívna – používa napr. algoritmus RR
- Fronta **procesov na pozadí** (background) – dávková – používa napr. FCFS
- Musí sa uplatniť vhodná politika, z ktorej fronty sa bude práve vyberať
 - prioritný výber – napr. dávková sa obsluhuje, len keď je interaktívna prázdna
 - **časové rezy** – napr. 80% času CPU pre interaktívne úlohy, 20% pre dávkové
- Možné aj iné rozdelenie front – na viac úrovní – procesy medzi nimi presúva plánovač
- Každá má svoj plánovací algoritmus a je definované, kedy preložiť proces do fronty s vyššou / nižšou preferenciou – zrenie procesu

6.) Fair Share Scheduler:

- Procesy sa delia do skupín, ktoré dohromady využívajú diel výkonu procesoru
- Plánovanie je prioritné a spravodlivé – priorita procesu klesá s rastom používania procesoru daným procesom alebo skupinou, do ktorej patrí
- Skupine, ktorej je pridelený väčší diel výkonu procesoru klesá váha pomalšie
- Pri exponenciálnom priemerovaní sa používa $\alpha = \frac{1}{2}$

Plánovanie homogénneho multiprocesoru (HMP):

- Jedna spoločná fronta pre všetky procesy alebo viac prioritne usporiadaných front
- Plánovacie politiky pri multiprocesore nemajú až taký význam - HMP umožňuje realizovať **zdieľanie záťaže** (load sharing) – každý procesor môže realizovať ktorýkoľvek sled - proces
- **SMP** – jedna centrálna fronta pripravených sledov
 - každý procesor si sám vyhľadáva nasledujúci sled
 - predbehnuté sledy pravdepodobne nebudú pokračovať na rovnakom procesore, preto nie je možné používať cache procesorov
 - sledy sa nemôžu spustiť zo spoločnej fronty paralelne
- **Gangy** – technika plánovania pre súčasný beh viac sledov 1 procesu na viacerých procesoroch
 - vhodné pre aplikácie, ktorých výkon klesá, ak sa neriešia paralelne

SYNCHRONIZÁCIA

Potreba IPC pri aktivitách, ktoré prebiehajú súbežne: multitasking, multithreading, multiprocessing, distribuované výpočty... Súbežné aktivity môžu:

- **súperiť** o zdroje (procesor, FAP, periférie) – procesom ich prideľuje OS
 - OS musí zamedziť **starnutiu** (resource starvation): procesu sú neustále odopierané zdroje, ktoré nevyhnutne potrebuje k svojmu dokončeniu
 - OS musí procesy izolovať, aby sa chybné neovplyvňovali, pretože súperiaci proces si „nie je vedomý“ existencie iných súperiacich procesov + zamedziť **uviaznutiu**
- **kooperovať** – vzájomne sa poznajú a zdieľajú istú množinu zdrojov, komunikujú medzi sebou
 - typicky kooperujú vlákna, ale aj procesy
 - výhody: zdieľanie zdrojov a informácií, eliminácia redundancie, paralelizmus

2 procesy môžu použiť **zdieľanú pamäť** pre výmenu dát medzi sebou – úloha **producent/konzument**:

- Jeden proces – producent – plní buffer (frontu) dátami; zvyšuje čítač fronty

- Druhý proces – konzument – dáta odoberá z fronty a operuje s nimi; znižuje čítač fronty
- Operácie counter++ a counter-- musia byť vykonané **atomicky** – bez prerušenia (viz. KS)
- Ak je fronta plná, producent musí čakať, kým z nej konzument niečo nezoberie
 - o producent sa dotazuje, či je vo fronte voľno
- Ak je fronta prázdna, konzument musí čakať, kým producent niečo vytvorí

Súbežný prístup k zdieľaným údajom môže spôsobiť, že dva rôzne procesy začnú operovať s jednou premennou (zdrojom) v tom istom čase – nutné zaviesť pravidlá – zamedziť „race condition“

- operácie zápisu musia byť vzájomne výlučné (buď sa zapíše jedna hodnota, alebo druhá)
- operácie zápisu musia byť vzájomne výlučné s operáciami čítania
- operácie čítania môžu byť realizované súbežne
- pre zabezpečenie integrity dát sa používajú **kritické sekcie**

Kritická sekcia – pasáž programu, v ktorej môže byť v danom okamihu najviac jeden proces

- Proces výlučne pristupuje ku zdieľaným zdrojom (napr. jednu premennú v danom okamihu modifikuje najviac jeden proces)
- **Podmienka bezpečnosti** (safety) = ak jeden proces vykonáva svoju kritickú sekciu, žiadny iný proces nemôže vykonávať svoju kritickú sekciu združenú s tým istým zdrojom
- **Podmienka živosti, postupu** (liveliness, progress) = ak žiadny proces nie je v KS združenej s istým zdrojom a existuje proces, ktorý chce vstúpiť do KS s týmto zdrojom, potom výber tohoto procesu sa nesmie odkladať nekonečne dlho
- **Podmienka spravodlivosti** (fairness) = ak proces požiada o vstup do kritickej sekcie, v konečnom čase mu musí byť vyhovie – môže ho predbehnúť najviac n procesov. Dovtedy je proces v stave **busy waiting** – v cykle sa neustále pýta, či už môže vstúpiť do kritickej sekcie / alebo čaká na signál

Elementárne riešenie: proces, ktorý vstúpi do KS zamaskuje prerušenie a pri odchode z KS zase povolí prerušenie – neefektívne, eliminuje rysy multiprogramovania, použiteľné len na 1-processorových systémoch (nezamaskuje prerušenie na iných procesoroch).

I.) Čisto SW riešenie (priame):

- Algoritmy, ktorých správnosť sa nespolieha na žiadne iné služby
- Používajú štandardný inštrukčný repertoár
- S aktívnym čakaním
- 1. riešenie: vytvoríme premennú, ktorú vidia oba procesy. Proces sa pozrie, či premenná obsahuje jeho číslo a ak áno, vstúpi do kritickej sekcie, a keď z nej bude odchádzať, dá tam číslo druhého procesu. Toto riešenie nesplňuje podmienku trvalosti postupu – napr. 1. proces ide do KS, skončí, nastaví v premennej číslo 2, ale 2. proces nechce ísť do KS. Po čase chce znova 1. proces vstúpiť do KS, ale v premennej je stále číslo 2 – uviazne.
- 2. riešenie: procesy dávajú žiadosť o vstup do KS nastavením príznaku – **flagu**. Ak má jeden proces zapnutý svoj flag, druhý čaká. Opäť nesplňuje podmienku trvalosti postupu: ak bežia paralelne a zapnú si flagy naraz, uviaznu.
- Petersnovo riešenie: kombinuje myšlienky 1. a 2., rieši predošlý problém - ak oba čakajú na vstup do KS (majú zapnuté flagy), paralelne sa pokúsia zapísať do zdieľanej premennej číslo toho druhého - ale jeden z týchto zápisov do pamäte nastane skôr, aj keď bežia paralelne – teda nestane sa, že oba budú čakať nekonečne dlho

- nesplňuje podmienku spravodlivosti, teda niektoré vlákno môže starnúť
- pri plne synchrónnom behu rozhoduje náhodná hodnota 0 alebo 1

II.) HW riešenie

- Pomocou špeciálnej inštrukcie procesoru – vhodné aj pre multiprocesory
- Používa sa inštrukcia **TSL, Test-and-Set Lock**, ktorá spôsobí:
 - načíta hodnotu premennej LOCK z operačnej pamäte do registru
 - a zároveň ju v pamäti nastaví na 1 – oboje atomicky, neprerušiteľne, v 1 cykle
- Ak v pamäti bola 1, prečítam 1 a znova zapíšem 1
- Ak tam ale bola 0, prečítam 0, ale v pamäti už bude 1
- Potom sa pozriem do registru: prečítal som 0? Ak áno, KS je voľná a môžem do nej vstúpiť
- Zároveň ostatné procesy vidia, že LOCK je už nastavený na 1
- Po ukončení kritickej sekcie proces zase nastaví LOCK na 0
- Podobné riešenie na iných procesoroch pomocou **XCHG** – zámena hodnôt register **x** pamäť
- Opäť nesplňuje podmienku spravodlivosti – ak dva procesy vykonajú naraz inštrukciu TSL, v pamäti sa tieto signály zaradia za seba, ale nedá sa predpovedať, v akom poradí
- Tak ako I.) riešenie je s aktívnym čakaním – teda spotrebovávajú čas procesoru
- Negatíva nevadia, ak sú kritickej sekcie krátke a riedko volané

III.) SW riešenie sprostredkované OS: Volanie služieb – *semafore, monitory, zasielanie správ*

Semafor (Dijkstra) – široko použiteľný synchronizačný nástroj

- Obecné: hodnota semaforu indikuje počet jednotiek zdroja, ktoré sú k dispozícii
- Je to špeciálny objekt: premenná typu integer + operácie vykonávané procesmi:
- **čakaj na udalosť** (wait, P) – ak je semafor zdvihnutý, tak ho zhoď (dekrementuj), ak nie je zdvihnutý (0 alebo záporná h.), čakaj na zdvihnutie semaforu vo fronte FIFO
- **oznám udalosť** (signal, V) – zdvihni (inkrementuj) semafor po opustení KS + odober proces z fronty čakajúcich procesov do fronty pripravených procesov
 - procesy sa nemusia neustále dotazovať na hodnotu semafora - *pasívne čakanie*
- Tieto operácie sa nazývajú aj **acquire** (P) a **release** (V)
- Binárny semafor (**mutex lock**, mutual exclusion) nadobúda hodnoty 0 alebo 1
 - Môžeme implementovať inštrukciami TSL alebo XCHG, alebo Petersnovým r.
- Obecný semafor môže nadobúdať ľubovoľnú celočíselnú hodnotu
 - Pre riadenie napr. 5 tlačiarň
 - Môžeme ho implementovať pomocou viacerých binárnych semaforov
 - Záporná hodnota indikuje, že žiadny zdroj nie je voľný – proces musí čakať
- Živosť a spravodlivosť je zaistená frontou FIFO, bezpečnosť zaistíme už z princípu semaforu
- Semafor sa môže používať aj ako synchronizátor – signalizuje vykonanie istej udalosti
 - Môže ale dôjsť k uviaznutiu: dva alebo viac procesov neobmedzene dlho čaká na udalosť, ktorú môže generovať len jeden s čakajúcich procesov
 - Môže dôjsť k starnutiu: proces sa nedostane von z fronty čakajúcich, ak ho predbiehajú procesy s vyššou prioritou
- Pomocou semaforu môžeme elegantne implementovať producent/konzument
 - inak nazývaný bounded-buffer problem – semafor mutex bude signalizovať, či proces môže pristúpiť k bufferu – len 1 producent alebo len 1 konzument
 - a ďalšie dva semafore pre indikáciu plného buffera (nesmie pristúpiť producent) alebo prázdneho buffera (nesmie pristúpiť konzument)

- Problémy so semaformi: sú príliš **nízkoúrovňové** (ako goto) – pri drobnej chybe sa môže stať, že povolíme vstup do KS; alebo zhadzujeme semafor a na konci práce znova zhodíme ten istý

Monitor – synchronizačný nástroj vysokej úrovne (vo VPJ)

- objekt, v ktorom sú deklarované procedúry, ktoré pracujú nad nejakou štruktúrou
- aby sa zamedzilo uviaznutiu, vykonávanie viacerých procedúr naraz *nepovolí kompilátor*
- v danom čase najviac jeden proces môže zavolať jedinu z procedúr monitoru – hovoríme, že **vstúpi do monitoru**
- aby proces mohol **čakať vnútri monitoru**, deklaruje sa premenná typu condition x;
 - o operácia x.wait(); - proces, ktorý ju vyvolá je blokový, až kým iný proces nevykoná
 - o operáciu x.signal(); - aktivuje práve jeden proces, ktorý čaká na podmienku x (ak žiadny proces nečaká na splnenie podmienky x, je to prázdna operácia)

Predávanie správ – mailboxy a porty

- **Mailbox** – schránka pre predávanie správ
- Vlastnia ju procesy: môže byť privátna pre 2 procesy alebo zdieľaná
- Proces ju môže zrušiť, alebo sama zaniká, keď končí proces
- **Port** – mailbox, ktorý patrí jednému prijímajúcemu procesu a viacerým procesom, ktoré zasielajú správy (napr. v modeli klient/server je prijímajúcim procesom server)
- Riešenie KS: procesy, ktoré sa chcú vzájomne vylučovať si budú posilať správu „go“
 - o do KS vstupuje proces, ktorý dostane správu „go“ ako prvý
- Producent/konzument pomocou mailboxu
 - o producent umiestňuje dáta do mailboxu *mayconsume*, pokiaľ získal správu z mailboxu *mayproduce*
 - o konzument získava dáta z mailboxu *mayconsume* a keď skončí, pošle správu do mailboxu *mayproduce*

Semafore, monitory aj predávanie správ sú **rovnako silné synchronizačné nástroje** – pre riešenie úloh môžeme VŽDY použiť ľubovoľný z nich, ale každý má inú filozofiu a iné výhody a nevýhody.

Problém čitateľa a zapisovateľa:

- Iná synchronizačná úloha ako producent/konzument
- Máme súbor, ktorého obsah môže meniť proces-pisateľ – v danom okamihu len jeden
- Jeho obsah môže čítať proces-čitateľ (čitateľov môže byť ľubovoľný počet)
- Ale operácia zápisu je exkluzívna
 - o writer nesmie začať zapisovať, pokiaľ je aktívny aspoň jeden reader
 - o počas doby zapisovania sa k súboru nesmie dostať žiadny reader ani iný writer
- a) s prioritou **čitateľa**: prvý čitateľ, ktorý pristúpi k zdroji zablokuje vstup pre pisateľov a *povolí vstup ostatným čitateľom* – posledný čitateľ, ktorý končí, uvoľní vstup pre pisateľa
- b) s prioritou **pisateľa**: počas zapisovania pisateľ zakáže prístup čitateľom – ak počas práce čitateľa požiada o vstup pisateľ, musí mu byť vyhovievané a čitateľa musia skončiť
- V prvom prípade starne pisateľ, v druhom čitateľ – pridáva sa podmienka, že *výlučný prístup k dátam pre daný proces vyprší po konečnom čase*

Problém večerajúcich filozofov:

- Pojednáva o násobnom získavaní viacerých prostriedkov do výlučného vlastníctva

- „5 filozofov sedí okolo okrúhleho stola, kde je misa špagiet a 5 vidličiek medzi každým párom. Každý filozof buď myslí alebo je ľavou a pravou vidličkou zároveň. Ak sa každý filozof chopí vidličky po svojej pravej ruke a potom bude čakať na vidličku vľavo, všetci zomrú od hladu.“
- Prístup ku každej vidličke bude sledovaný semaforom mutexom – vidlička je voľná alebo nie
- Filozof získa pravú vidličku, potom ľavú vidličku, je, vráti vidličky a zase myslí
- Možné riešenia:
 - o Zrušenie symetrie – jeden filozof bude ľavák
 - o Semaforey: v jedálni s jedlom pre n filozofov bude len n-1 stoličiek – jeden čaká von
 - o Filozof sa smie chopiť vidličky, len keď sú obe voľné („nie sú v kritickej sekcii“)
 - Riešenie pomocou monitoru – filozof f čaká na splnenie podmienky, že obaja jeho susedia nejedia
 - Než začne jesť, zavolá procedúru pickUp(f), ktorá sa dokončí, keď bude splnená podmienka
 - Keď doje, alebo jeho susedia čakajú na vidličky, zavolá procedúru putDown(f)

UVIAZNUTIE



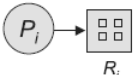
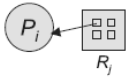
Množina procesov P uviazla, keď každý proces z P čaká na udalosť, ktorú vyvolá len niektorý z ďalších procesov P. Uviaznutie môže nastať, keď súčasne platia tieto 4 podmienky:

- **vzájomné vylúčenie:** zdieľaný zdroj môže v danom okamihu používať len jeden proces
- **požiadavky sa uplatňujú postupne:** proces vlastníci aspoň 1 zdroj čaká na získanie ďalšieho zdroja vlastneného iným procesom (procesy si držia zdroje a čakajú, kto uvoľní svoje)
- **nepripúšťa sa predbiehanie:** zdroj môže uvoľniť len proces, ktorý ho v danej dobe vlastní
- postačujúca, nie nutná podmienka: došlo k zacykleniu požiadaviek

Model skúmania uviaznutia: zdroje (čas CPU, priestor pamäti, I/O zariadenia)

- majú rôzne inštancie
- sú používané podľa nasledujúcej schémy: žiadosť o pridelenie zdroja (request), použitie zdroja, uvoľnenie zdroja (release)

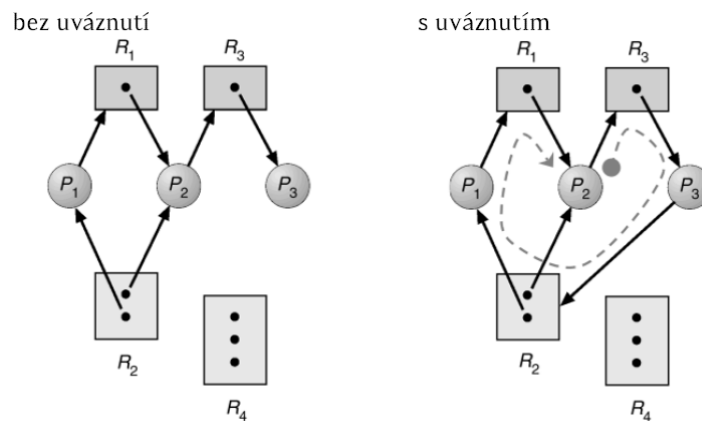
RAG: Resource Allocation Graph, graf pridelenia zdrojov:

- množina uzlov V sa delí na dva typy:
 - o P = množina procesov v systéme: 
 - o R = množina zdrojov v systéme (+ počet inštancií): 
- hrana **požiadavky**: orientovaná hrana $P_i \rightarrow R_j$ 
- hrana **pridelenia**: orientovaná hrana $R_i \rightarrow P_j$ 

Ak v grafe nie je cyklus, k uviaznutiu nedošlo.

Ak v grafe je cyklus

- a existuje jediná inštancia zdroja daného typu, k uviaznutiu došlo.
- a existuje viac inštancií zdroja daného typu, k uviaznutiu môže dôjsť.



Ako zvládnuť uviaznutie:

- **prevencia:** už pri návrhu zabrániť tomu, aby nastali podmienky (viz. vyššie), ktoré by dostali systém do stavu uviaznutia
- **obchádzanie:** platnosti nutných podmienok uviaznutia sa zamedzuje za behu, napr. prostriedok sa nepridelí, ak by hrozilo uviaznutie
- **detekcia** a obnova stavu po uviaznutí
- **ignorovať** – abort procesu – používa väčšina OS

PREVENCIA: konzervatívna politika

- nepriame metódy:
 - o nepoužívanie zdieľaných zdrojov – zneplatnenie podmienky vzájomnej výlučnosti, virtualizácia prostriedkov (spooling)
 - o keď proces niečo požaduje, nesmie vlastniť žiadny iný prostriedok – musí požiadať o všetky prostriedky naraz pri svojom vytvorení, alebo môže žiadať len keď nič nevlastní – zneplatnenie podmienky postupného uplatňovania požiadaviek (ale nízka efektivita, starnutie)
 - o odoberanie prostriedkov správcom: uvoľní dosiaľ držané prostriedky pred žiadosťou o ďalší *alebo* procesy umiestňuje do čakacích front a aktivuje ich, keď bude daný zdroj k dispozícii – zneplatnenie podmienky zabránenia predbiehania
- priama metóda:
 - o zabránenie zacyklenia v grafe definovaním poradia prideľovaných prostriedkov

OBCHÁDZANIE: dynamické rozhodovanie o pridelení zdroja počas behu procesu. Systém musí mať nejaké dodatočné informácie *a priori* – napr. proces deklaruje maximálnu počet prostriedkov každého typu, ktoré bude požadovať.

- Algoritmus riešiaci obchádzanie uviaznutia skúša, či pridelenie zdroja nespôsobí uviaznutie = tzn., či ponechá systém v bezpečnom stave (**safe state**). Ak by systém prešiel do unsafe state, prechod do stavu uviaznutia je hrozbou.
- Postupnosť procesov je **bezpečná**, ak požiadavky každého je možné uspokojiť práve voľnými zdrojmi a zdrojmi držanými ostatnými procesmi (vtedy sa čaká).

RAG:

- nároková hrana $P_i \rightarrow R_j$ indikuje, že proces P_i môže niekedy v budúcnosti požadovať zdroj R_j (vedie rovnakým smerom ako požiadavková hrana, ale je zobrazená čiarkovane)

- konvertuje sa na požiadavkovú hranu v okamihu, keď proces P_i požiadá o zdroj R_j
- keď proces zdroj získa, požiadavková hrana sa mení na hranu pridelenia
- keď proces zdroj uvoľní, hrana pridelenia sa mení naspäť na nárokovú hranu
- nesmie vzniknúť cyklus

Bankárov algoritmus – Dijkstra – bankár má vopred peniaze (prostriedky) a jeho zákazníci vopred deklarujú maximálnu výšku úveru (maximum prostriedkov), ktorú budú požadovať. Úvery splatia (prostriedky uvoľnia) v konečnom čase. Bankár nemôže poskytnúť úver, ak si nie je istý, že môže nejakým spôsobom uspokojiť v konečnom čase všetky požiadavky ostatných zákazníkov.

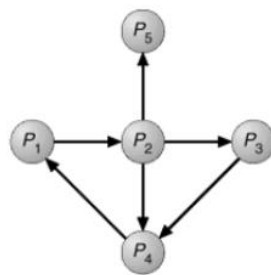
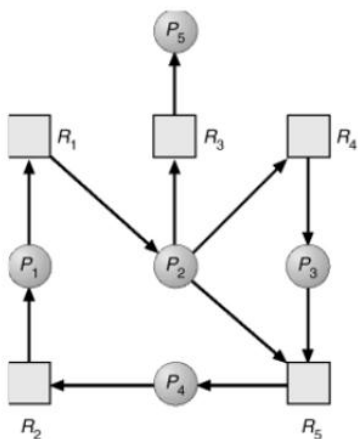
- Proces požadujúci pridelenie môže byť daný do stavu čakania.
- Predpokladá sa najhorší prípad - všetky procesy si vyžadujú deklarované maximá.

Implementácia: viz príklady: 32, 35-39

- n – počet procesov, m – počet typov zdrojov
- available = vektor dĺžky m , indikuje počet dostupných inštancií daného zdroja
- max = matica $n \times m$, na každom riadku je vektor vopred deklarovaných maxim procesu
- allocation = matica $n \times m$, indikuje, koľko inštancií daného zdroja má práve proces pridelený
- need = matica $n \times m$, obsahuje vektory možných požiadavkov procesu; koľko ešte môže vyžadovať (max – allocation)
- request = vektor okamžitých požiadavkov procesu

DETEKCIA: umožňuje sa, aby systém uviazol a následne sa aplikuje plán obnovy – násilne sa ukončujú jednotlivé procesy. (Príklad: 46-47)

Graf čakania: uzly sú procesy, hrana $P_i \rightarrow P_j$, ak P_i čaká na P_j .



Vždy, keď nie je možné uspokojiť požiadavku na pridelenie, algoritmus v grafe čakania hľadá cykly = detekuje sa proces spôsobujúci uviaznutie a uviaznuté procesy.

Niekedy nie je možné určiť, ktorý z mnohých procesov spôsobil uviaznutie – algoritmus detekcie uviaznutia sa vyvoláva náhodne.

Plán obnovy: násilné ukončenie uviaznutých procesov; alebo sa ukončujú jednotlivé procesy, pokiaľ sa neodstráni cyklus. Poradie násilného ukončenia môžu udávať rôzne kritériá: priorita procesu, doba behu procesu, doba potrebná k ukončeniu procesu, prostriedky, ktoré proces použil... Môže dôjsť k starnutiu – niektorý proces bude vybraný ako obeť trvale.

SPRÁVA OPERAČNEJ PAMÄTE

Program riadiaci beh procesu musí byť umiestnený v operačnej pamäti – na aké miesto, o tom rozhoduje **správa pamäte**. Správa pamäte je predmetom činnosti OS. Musí zaistiť:

- Možnosť **relokácie** programov – ich premiestňovanie v pamäti

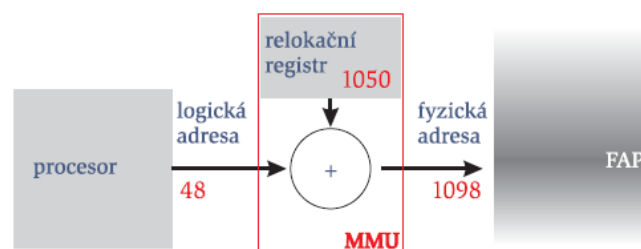
- Programátor nemôže vedieť, kde bude jeho program umiestnený
- Po swap-out a následnom swap-in sa pozícia programu v pamäti môže zmeniť
- Ochrana: proces môže siahť len na pamäťové miesta, ktoré mu patria
- Logickú organizáciu: programy sú moduly so vzájomne odlišnými vlastnosťami
 - Moduly s inštrukciami sú väčšinou execute-only
 - Moduly s dátami sú buď read-only alebo read/write
 - Moduly môžu byť privátne a verejné
- Možnosť zdieľania: viac procesov môže zdieľať spoločnú časť pamäte
- Efektivitu

Adresový priestor, AP: škála adres buniek hlavnej pamäte; každý proces má svoj vlastný AP => hypotetické delenie (nie presne definované hranice) pamäti na úseky patriace procesom – **LAP**.

Adresa v LAP = logická, virtuálna. Adresa vo FAP (pole buniek pamäte) = fyzická, reálna.

Viazanie LAP na FAP je základový koncept správy pamäte:

- **Statické (pri kompilácii)** - viazanie LAP na FAP identitou
 - umiestnenie programu vo FAP je známe vopred, pred prekladom (vstavané systémy)
 - pri kompilácii vzniká **absolútny program** - nepredpokladá sa výmena dát v pamäti
 - pri zmene umiestnenia programu vo FAP sa musí preklad opakovať
- **Statické (pri linkovaní)** - opäť LAP = FAP
 - umiestnenie programu vo FAP je známe pri zostavovaní programu
 - prekladač generuje **zostavovateľný modul**, pri linkovaní sa zobrazuje z LAP do FAP
- **Dynamické** = viazanie adres LAP na FAP až za behu – pri interpretácii inštrukcie
 - proces môže meniť svoju polohu vo FAP medzi rôznymi fázami behu
 - musí byť HW podpora – *Memory Management Unit, Dynamic Address Translation*
 - najjednoduchšie riešenie – práca s **relokačným** (bázovým) **registrom** CPU
 - nastaví sa na počiatočnú hodnotu, o ktorú sa bude posúvať logická adresa
 - keď je táto adresa použitá ako ukazateľ do hlavnej pamäti, obsah relokačného registru sa pripočítava k adresám interpretovaných inštrukcií
 - relokačný register je privilegovaný, dostupný len OS



- spolu so **swapovaním** a **prekryvmi** (v pamäti sa nechávajú len tie úseky, ktoré „budeme najskôr potrebovať“) historická technika, používa sa už len napr. v RT-OS
- moderné riešenie: stránkovanie/segmentovanie, t.j. **virtualizácia pamäte** (viz. ďalej) + pri programovaní – DLL (knížnica sa zavolá, až keď ju bude treba)

Správa pamäti:

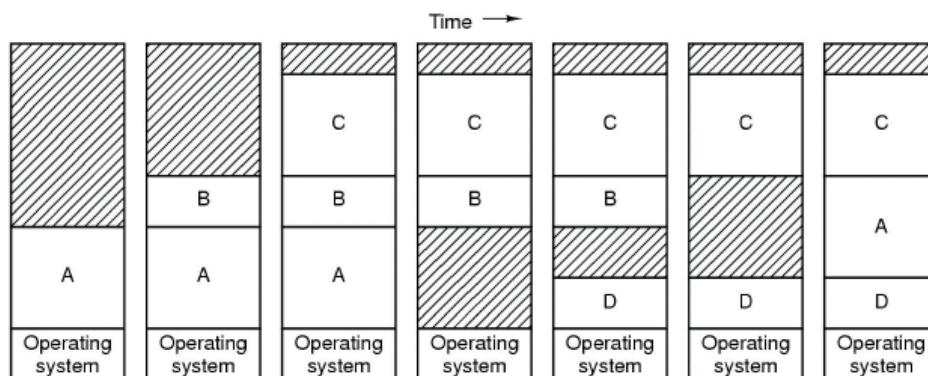
- FAP zdieľa OS a jeden aktuálne bežiaci program (monoprogramovanie)
- FAP zdieľa OS a n procesov (historické dávkové systémy)
- FAP sa rozdelí do dvoch typov sekcií – 1.) na začiatku je OS, 2.) sekcie pre užívateľské procesy

- pre ochranu OS a užívateľských procesov alebo procesov medzi sebou je možné použiť relokačný register (bude adresy posúvať o toľko, koľko zaberá OS) a **medzný register** (najvyššia dostupná adresa)

Metóda pridelenia súvislých oblastí: procesu sa prideli sekcia voľnej pamäti vo FAP, ktorá uspokojí jeho požiadavku. Pre pridelenie sekcie – rôzne techniky: **first-fit, best-fit, worst-fit**.

Vznikajú ale úseky voľného miesta roztrúsené po FAP – nastáva:

- **externá fragmentácia** – súhrn voľnej pamäti je dostatočne veľký, ale nie je to súvislý celok
- **interná fragmentácia** – pridelená oblasť pamäti je priveľká, zvyšok sa nevyužije
- Presun blokov – dodatočná réžia, nutná relokácia (MMU)



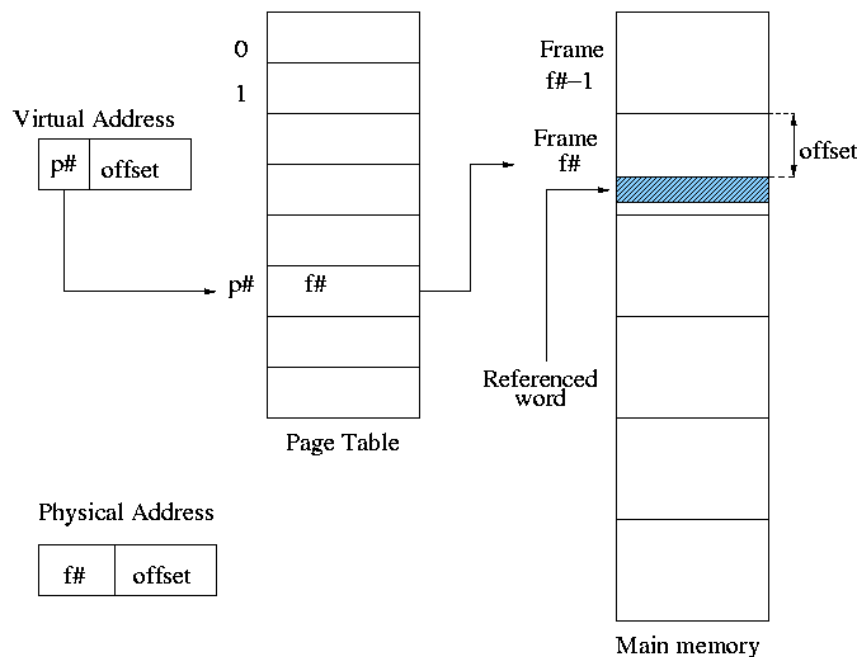
STRÁNKOVANIE – myšlienka: Proces v danom okamihu potrebuje len obmedzenú časť svojho AP.

Nech sa LAP aj FAP logicky rozdelí (nie presne definované hranice) a potrebný text programu nech sa rozmiestni do pamäti po častiach (nemusia byť súvislé).

- FAP (reálna, fyzická pamäť) sa rozdelí na **rámce, frames**
- LAP sa rozdelí na **stránky, pages**
- Obe majú zhodnú pevnú dĺžku (napr. 512 B)
- LAP nie je zobrazovaný do jednej súvislej sekcie FAP, ale do viacerých častí, rámcov FAP
- Poradie pridelených rámcov vo FAP nijako nesúvisí s poradím stránok v LAP

Keď proces vyžiada prístup ku svojej pamäti, poskytne OS logickú adresu. Táto logická adresa je index do **tabuľky stránok** (Page Table), v ktorej sú uložené mapovania logických adries na fyzické.

- Záznam v tabuľke pre stránku p obsahuje číslo rámca f , ktorý obsahuje stránku p
- Tabuľka stránok teda realizuje **preklad** – „kde vo fyzickej pamäti leží konkrétna stránka?“
- Obsah PT nastavuje OS
- Každý proces má vlastnú PT – musí sa dynamicky prepisovať pri prepnutí kontextu



- PT je uložená v hlavnej pamäti
- Je odkazovaná registrom **PTBR** (Page Table Base Register)
- Sprístupňovanie údajov v tabuľke stránok sú operácie navyše – zníženie efektivity sa rieši špeciálnou rýchlou HW cache pamäťou – **TLB** (Translation Look-aside Buffer)
 - o asociatívna pamäť, obsahuje k posledne použitých dvojíc $\{p, f\}$
- Preklad $p \rightarrow f$: ak je p v TLB, získa sa hodnota f odtiaľ, inak sa pristúpi do PT
- Počítame skutočnú dobu prístupu do pamäte: **EAT – Effective Access Time**
 - o sprístupnenie TLB = ϵ časových jednotiek (napr. nanosekúnd, ns)
 - o prístup k hlavnej pamäti = t ns
 - o hit ratio (P, že číslo stránky bude v TLB) = α
 - o $EAT_{TLB} = (TLB + t) * \alpha + (TLB + PT + t) * (1 - \alpha) = (\epsilon + t) * \alpha + (\epsilon + 2t) * (1 - \alpha) = (2 - \alpha) * t + \epsilon$
 - o Pr.: $\epsilon = 20$ ns, $\alpha = 80\%$, $t = 100$ ns $\rightarrow EAT_{TLB} = (2 - 0,8) * 100 + 20 = 140$ ns

Iné typy tabuliek stránok:

- **Viacúrovňová**: ak používame krátke stránky – lepšia manipulácia, ale veľa krátkych stránok = obrovská PT, preto sa radšej používa rozloženie tabuliek do viacerých úrovní (typicky 2 – stránkový adresár obsahuje ukazatele na viaceré tabuľky stránok – pretože pre každú úroveň sa musí siahť do pamäte)
- **Invertovaná**: obsahuje pevne toľko riadkov, koľko je rámcov vo FAP. Funguje ako asociatívna pamäť: v každom riadku je pid (process id, aby sa vedelo, akému procesu patrí daná oblasť v pamäti), číslo logickej stránky, a číslo fyzickej stránky (nie priama adresa do FAP)
- **Hašovaná**: mapovanie virtuálnej adresy do indexu napr. invertovanej tabuľky stránok

Zdieľanie stránok: príklad - spustím si 3x textový editor a v každom okne pracujem s inými dátami. Stačí ale, aby bol samotný program umiestnený vo FAP raz – inštrukcie sú stále rovnaké, každý proces má vyhradené samostatné stránky len pre dáta.

SEGMENTÁCIA – program je kolekcia samostatných *segmentov* (napr. 1 segment = 1 procedúra); každý segment má programátorom priradenú rolu. Pri spustení programu sa tieto (rôzne veľké) segmenty umiestňujú do FAP = **delenie pamäti na segmenty**. V danej chvíli sa využíva len istá časť.

Keď sa proces odkazuje na segment, poskytuje logickú adresu = dvojica {číslo segmentu s , offset d }. Transformácia LAP → FAP pomocou **segmentovej tabuľky** (ST), ktorá uchováva záznamy o každom segmente v pamäti – jeho počiatočnú adresu (*bázu*) a celkovú dĺžku (*limit*).

Pomocou čísla segmentu s sa v tabuľke vyhladá báza konkrétneho segmentu vo FAP, offset d určuje posun v rámci jedného segmentu (+ kontrola, či nepresiahol limit).

V čistej podobe sa už nepoužíva, pretože segmenty potrebujú súvislé oblasti pamäte, ale **kombinuje sa so stránkovaním** – ku každému segmentu prislúcha tabuľka stránok. V hlavnej pamäti sa potom uchovávajú len PT zavedených segmentov. ST už neobsahuje bázu segmentu, ale adresu jeho PT.

- 32 bitová adresa v segmente => veľkosť segmentu = 4 GB
- 16 K (16384) segmentov => veľkosť LAP = 16 K * 4 GB = 64 TB

VIRTUÁLNA PAMÄŤ

Proces v danom okamihu potrebuje len časť svojho LAP – bezprostredne nutných je len niekoľko inštrukcií, ktoré sa majú vykonať. LAP je manipulovaný po stránkach – časť LAP procesu je zobrazená v **reálnej pamäti** (FAP), celý LAP procesu je zobrazený na **disku**. Keďže LAP je podstatne väčší než FAP - všetky stránky sa naraz nevojdú do operačnej pamäte - implementuje sa *virtuálna pamäť*.

Dôvody pre virtualizáciu pamäte:

- Preklad adres LAP do FAP sa robí čo najneskôr, najlepšie za behu procesu
- Pridelovanie súvislých oblastí FAP procesu spôsobuje externú fragmentáciu (napr. dám mu 4 MB, ale on potrebuje len 1 MB a zvyšné 3 nemôže nikto využiť) – stránkovanie to eliminuje
- Stupeň multitaskingu má byť čo najvyšší, tým ale rastú nároky procesov na priestor vo FAP
- Existujú procesy, ktoré majú vyššie pamäťové nároky ako je celý rozsah FAP
- Pri behu procesu sa nemusí vo FAP naraz uchovávať celý program a všetky potrebné dáta

Techniky: **stránkovanie** na žiadosť, demand paging,
segmentovanie na žiadosť, demand segmentation.

OS pri štarte procesu zavedie do FAP len malú časť programu (prvých pár stránok jeho LAP). Proces **odkazuje**, referencuje na stránky svojho LAP:

- *legálne* = stránka sa nachádza vo FAP, procesu je dostupná, LAP sa preloží na FAP
- *nelegálne* = ak **a)** stránka nepatrí procesu => abort, **b)** stránka nie je práve v operačnej pamäti, ale je explicitne požadovaná = je potrebná – jej zavedenie sprostredkuje OS

Pre preklad logickej adresy na fyzickú (kde v pamäti sa nachádza konkrétna stránka?) sa používa tabuľka stránok, **PT** (prípadne ST pri segmentácii).

- Tabuľka nie je na pevnom mieste – existuje register CPU: ukazateľ na tabuľku stránok
- Každá položka PT obsahuje bit **Present**, resp. **Valid/Invalid** (V/I) indikujúci zavedenie odpovedajúcej stránky (segmentu) do FAP (1 = je vo FAP, 0 = nie je)

Časť procesu vo FAP nazývame **rezidentnou množinou** (resident set). Odkaz mimo rezidentnú množinu (= na stránku, ktorá nie je vo FAP = na stránku, ktorá má bit V/I na 0) spôsobuje **prerušenie** typu **výpadok stránky** (segmentu) – Page/Segment Fault.

- OS označí takýto proces za čakajúci (wait) na stránku/segment vo FAP

- OS vyhľadá stránku vo vonkajšej pamäti a nahrá jej obraz do voľného rámca vo FAP (**DMA**)
- Proces zatiaľ čaká - na CPU môžu bežať iné procesy
- Ak existuje voľný rámec, použije sa, ak nie, musí sa vybrať rámec, ktorý sa uvoľní
- OS upraví bit V/I na 1 - stránka už je v operačnej pamäti
- Generuje sa prerušenie a proces sa zaradí medzi pripravené (ready)

Fetch policy - kedy stránku zavádzať do FAP?

- Na žiadosť, keď na ňu proces odkazuje a nie je prítomná vo FAP
- Predstránkovať – keďže susedné stránky v LAP sa pravdepodobne budú používať v krátkom čase po sebe, zavediem do FAP vopred niekoľko susedných stránok (keď beriem stránku 10, tak aj 11, 12, ...) – aj keď ich možno nevyužijem, zabránim neskorším výpadkom. Obzvlášť vhodné pri inicializácii procesu

Placement policy - kam stránku umiestniť vo FAP?

- Stránkovanie: ľubovoľný voľný rámec
- Segmentácia: first-fit, best-fit, worst-fit, ...
- Stránkované segmenty: ľubovoľný voľný rámec

Najdôležitejšia: **Replacement policy** – ktorú stránku **nahradiť**?

- FAP je plná a nie je dostupný voľný rámec
- OS nevie nič o bežiacich procesoch - musí vybrať, ktorú stránku odoberie ktorému procesu
- **Lokálny** výber, intro = obeť sa hľadajú medzi stránkami procesu, ktorý vyvolal page fault
- **Globálny** výber, extra = obeť sa hľadajú medzi stránkami všetkých procesov
- Ak sa do stránky nezapísalo, môžeme ju rýchlo vyhodiť, lebo jej kópia je na disku
- Ak sa do stránky zapísalo, pred vyhodením ju musíme zapísať na disk – pomalšie
- Súčasťou stránky je bit **Dirty** – ak sa do nej aspoň raz zapísalo, nastaví sa na 1

Rôzne algoritmy pre **výber obete** – ideálny, optimálny algoritmus = vyhodí sa *stránka, ktorá bude najneskôr odkazovaná* – to sa nedá reálne zistiť, používa sa pre zrovnávanie iných algoritmov:

- **FIFO** – obeť = stránka, ktorá je najdlhšie zobrazená vo FAP
 - o nedá sa reálne použiť - prvá nahraná stránka obsahuje väčšinou dôležité dáta
- **LRU** – obeť = stránka, na ktorú sa najdlhšie nikto neodkázal
 - o blízko optima, ale náročná a neefektívna implementácia
 - o napr. čítače behu času, mikroprogram pričítava +1 po nejakom časovom úseku – spotreba CPU a problém pretečenia, preto sa hľadajú algoritmy aproximujúce LRU:
- **Bity referencie** – slabika (1B) pri každej stránke v PT, ilustruje 8 posledných krokov
 - o iniciálne 00000000 – po prvej referencii: 00000001, po druhej: 00000011...
 - o problém: stránka sa nahrá, ale proces je ešte čakajúci, vidí sa, že stránku nikto nereferencuje, a kým bude proces spustený, stránka sa zase vyhodí
- **Druhá šanca** – každá stránka má navyše jeden bit (extra život) – ak je označená za obeť (napr. pomocou FIFO), bit sa nuluje a hľadá sa iná obeť – vyhodí sa až stránka s nulovým bitom
 - o vylepšená verzia: pridáva sa bit modifikácie (dirty bit)
- **LFU/MFU** – najmenej/najviac často referencovaná stránka

Prideľovanie rámcov:

- **Pevné** = pamäť sa rozdelí *fixne* na úseky s rovnakým počtom rámcov alebo *proporcionálne* – podľa veľkostí LAP jednotlivých procesov = neefektívne + nemožnosť využitia globálnych obetí pri uvoľňovaní rámca (procesu patrí istý počet rámcov a ak mu nestačia, vyhodnotením rámcov, ktoré patria inému procesu nič nezíska, lebo by k nim nemohol prístupit)
- **Prioritné** = počet rámcov pridelených procesu sa v čase dynamicky mení
 - o globálne náhrady – najjednoduchšie implementovateľné, používa sa v mnohých OS, ale nebezpečenstvo **thrashingu** – veľa procesov má malý počet rámcov, generujú výpadky veľmi rýchlo – CPU nerobí nič iné len vymieňa stránky – slabý výkon
 - o lokálne náhrady – pre daný proces sa vytvorí tzv. **pracovná množina** posledných referencovaných stránok v danom **okne** (ktoré má pevnú dĺžku). Keď je v okne malý počet stránok (v istom čase stále referencuje niekoľko dookola), tak proces generuje málo page faultov (má **nízkú frekvenciu výpadkov**) a odoberajú sa mu rámce. Keď začne generovať veľa výpadkov, pridávajú sa mu rámce, lebo ich potrebuje.

Problémy s veľkosťou stránky:

- Malá = malá vnútorná fragmentácia, málo výpadkov, ale veľká PT a veľa diskových operácií.
- Veľká stránka = obsahuje veľa dát => je často potrebná = časté výpadky. Tak či tak silná degradácia výkonu za cenu multitaskingu.

Stránkovanie - paging	Segmentácia - segmentation
FAP je 1-dimenzionálny, rozdelený na rámce pevnej dĺžky	FAP je 1-dimenzionálny, rozdelený na segmenty premenlivej dĺžky
LAP je 1-dimenzionálny, rozdelený na stránky s rovnakou dĺžkou ako majú rámce	LAP je 2-dimenzionálny, logická adresa = číslo segmentu + offset
Interná fragmentácia v rámcoch, žiadna externá fragmentácia	Žiadna interná fragmentácia (pridelí sa práve toľko, aký veľký je segment), externá fragment.
OS udržiava tabuľku stránok, každá stránka odkazuje na rámec	OS udržiava tabuľku segmentov s bázou a dĺžkou každého segmentu
OS udržiava tabuľku rámcov definujúcu stav každého rámca (či je voľný)	OS udržiava zoznam voľných oblastí vo FAP
LAP → FAP sa rieši dynamicky, preklad: PT	LAP → FAP sa rieši dynamicky, preklad: ST
Stránky procesu sa zavádzajú podľa potreby	Segmenty procesu sa zavádzajú podľa potreby
Zavedenie jednej stránky do FAP môže spôsobiť výpis (vyhodnenie) jednej stránky na disk	Zavedenie jedného segmentu do FAP môže spôsobiť výpis viacerých segmentov na disk

PODSYSTÉM VSTUPU A VÝSTUPU: Existuje mnoho rôznych I/O zariadení s rôznymi vlastnosťami a spôsobmi ovládania, ktoré OS musí vedieť spravovať.

Základné spoločné rysy z hľadiska architektúry:

- **Port** = prípojné adresovateľné miesto periférie, cez ktoré sa prenášajú informácie
- **Zbernica** = subsystém pre prenos čítaných/zapisovaných dát z/do periférie
- **Radič** (adapter) = rozhranie (interface) periférie na CPU – prijíma príkazy/dáta z CPU a dodáva dáta/indikácie do CPU + dáva podnet pre generovanie prerušenia

I/O inštrukcie strojového jazyka – čítanie/zápis z/do portu, zadávanie príkazov I/O zariadeniu, odoberanie stavových informácií od I/O zariadenia.

- I/O adresy sú mapované do FAP – inštrukcie LOAD, STORE, MOV
- I/O adresy sú mapované do samostatného I/O priestoru - inštrukcie IN, OUT

- Definujeme dátové štruktúry: *binary-semaphore S1, S2, int C;*
- Inicializácia: *S1 = 1; S2 = 0; C = iniciálna hodnota semaforu S;*
- Operácie **acquire:**

```
acquire(S1);  
if (C < 0) {  
    release(S1);  
    acquire(S2);  
}  
release(S1);
```

a

release:

```
acquire(S1);  
C++;  
if (C <= 0) release(S2);  
else release(S1);
```