Operačné systémy / Správa pamäte

Dušan Bernát
bernat@fmph.uniba.sk

Prehľad

- História a motivácia, monoprogramovanie,
- úseky pevnej a premenlivej dĺžky, fragmentácia,
- stránkovanie, segmentácia,
- logický adresový priestor procesu, preklad logickej adresy na fyzickú,
- tabuľka stránok a jej realizácia,
- výpadok stránky,
- algoritmy výberu obete,
- swapovanie,
- page cache/buffer cache,
- malloc, overcommit, OOM killer,
- hierarchický pamäťový systém, princípy lokality.

Pamäť bez swapovania

Monoprogramovanie

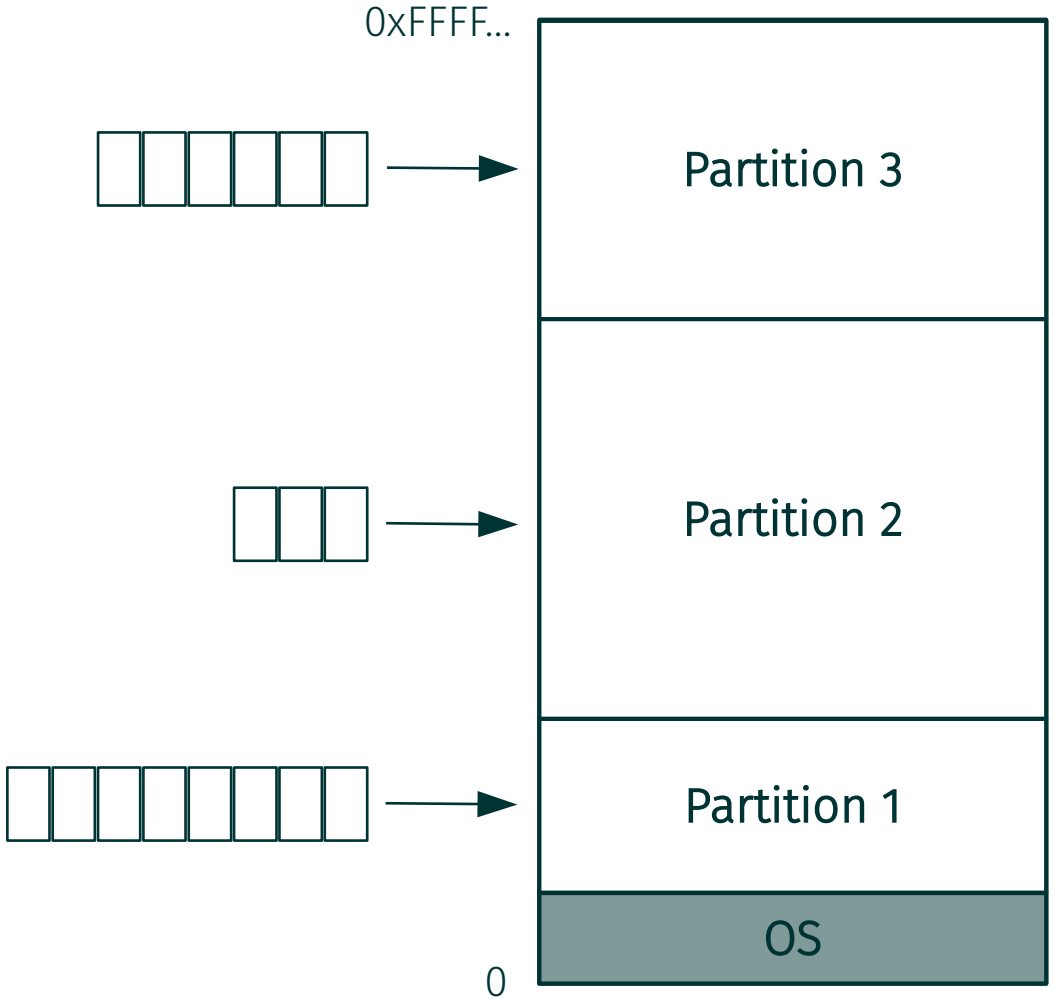
- Pôvodne bol v pamäti len jeden program – monoprogramovanie.
  - Celú (fyzickú) pamäť využíval len jeden program a operačný systém (ROM).
  - Najjednoduchšia forma správy pamäte.
- Nerobili sa žiadne výmeny (*swapping*) obsahu na sekundárne médium.
  - Pokiaľ sa program do pamäte nezmestil, programátor ho mohol rozdeliť (*overlay*).
- Každý program sa zavádzal na tú istú adresu (relokácia nebola potrebná).
- Ochrana pamäte OS – zavedenie OS s každou úlohou, OS v ROM, podpora HW.
- Napríklad: kedysi MS DOS, dnes možno jednoduché vnorené systémy.
- Program zavedený do pamäte sa vyonával až kým sa neukončil, alebo kým nebol zablokovaný vstupno-výstupnou operáciou.
  - Procesor bol teda počas čakania na dokončenie V/V nevyužitý.
- Snaha o umiestnenie viacerých programov do pamäte bola motivovaná zvyšovaním využitia procesora.

Multiprogramovanie

- Pamäť bola pri štarte OS rozdelená na viacero **úsekov pevnej dĺžky** (segmenty), mohli byť aj rôzne veľké. Ich počet sa nemenil (len pri štarte).
- Novému procesu sa pridelí najmenší vhodný voľný úsek.
- Problémom je **interná fragmentácia** – program nemusí využiť všetku pridelenú pamäť. Táto časť pamäte je teda nevyužitá, ale alokovaná.
- **Relokácia** je potrebná v čase zavádzania (preklad adries, alebo nastavenie bázového registra na začiatok prideleného úseku).
- **Ochrana** pred vzájomným prístupom môže byť realizovaná registrom pre limit.
- Každý úsek môže mať vlastný rad čakajúcich programov. Alebo môže byť jeden spoločný rad a ak nie je voľný menší úsek, môže sa prideliť aj väčší.
- Z pohľadu jedného úseku pamäte sa využívalo stále monoprogramovanie.
- Napríklad: IBM System/360 (MFT – Multiprogramming with Fixed number of Tasks).

Multiprogramovanie

- Nový proces sa pridá do radu podľa požadovanej veľkosti.
- Jednotlivé rady sú obsluhované FCFS.
- Procesorový čas sa delí medzi úlohy v pamäti.



Príklad: Multiprogramovanie

- Nech každý proces v priemere polovicu času čaká na vstupno-výstupnú operáciu. Ako sa zmení využitie procesora, ak budú v systéme štyri procesy?

Príklad: Multiprogramovanie

- Nech každý proces v priemere polovicu času čaká na vstupno-výstupnú operáciu. Ako sa zmení využitie procesora, ak budú v systéme štyri procesy?
- Pravdepodobnosť čakania na V/V: $p = 0.5$
- Využitie procesora – pravdepodobnosť, že procesor niečo robí ($u = 1 - p$).
- Pre jeden program: $\boldsymbol{u_1 = 1 - p = 0.5}$
- Pravdepodobnosť, že všetkých n procesov čaká súčasne: $p^n = 0.5^4 = 1/16$
- Využitie pri n procesoch: $\boldsymbol{u_n = 1 - p^n = 1 - 1/16 = 0.938}$
- Využitie procesora pri jednom procese je *50%* a pri štyroch procesoch bude *93.8%*.
- Prakticky to bude menej, lebo napríklad aj OS spotrebuje nejaký čas na prepínanie procesov.

Pamäť so swapovaním

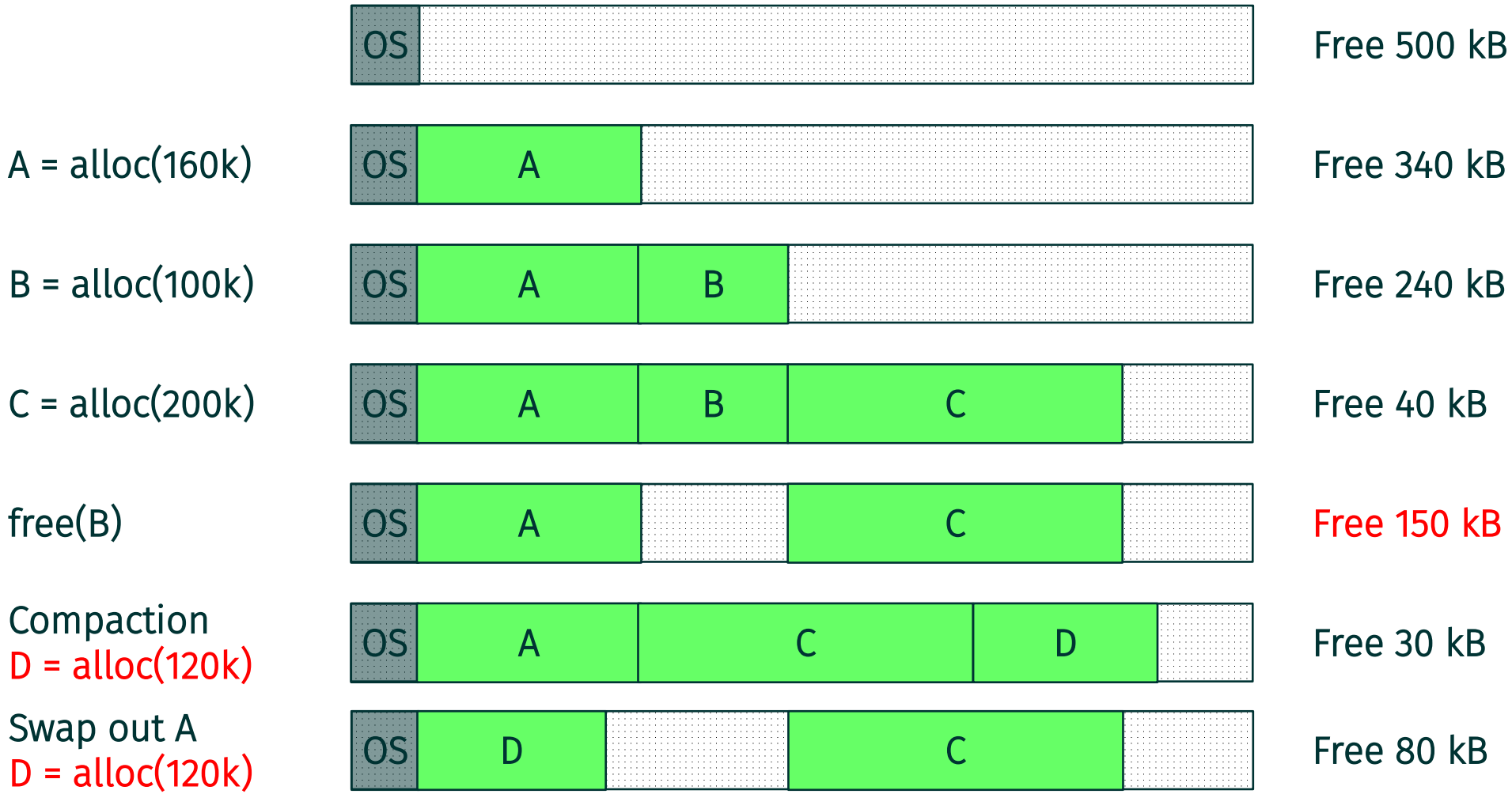
Multiprogramovanie s úsekmi premenlivej dĺžky

- Presúvanie celého procesu medzi hlavnou pamäťou a diskom sa nazýva výmena (*swapovanie*).
- Úseky prementlivej dĺžky – veľkosť a teda aj počet jednotlivých úsekov pamäte sa za behu mení. Umiestnenie segmentu tiež.
- Virtuálne aj fyzické adresy úseku sú súvislé (jeden blok).
- Program však môže byť presunutý na iné miesto. Potrebná je **relokácia** za behu.
- Interná fragmentácia je premenlivou dĺžkou pridelovaných úsekov eliminovaná.
- Voľná pamäť sa postupne rozdrobí uvoľnenými úsekmi.
- Nevyužité (malé) úseky pamäte medzi pridelenými úsekmi – **externá fragmentácia**.
- Napríklad: IBM OS 360 MVT (Multiprogramming with Varying number of Tasks), PDP-10 OS.

Externá fragmentácia

- Voľná pamäť je rozdrobená na malé časti medzi obsadenými úsekmi.
- Nevyužiteľná voľná pamäť je mimo alokovaných oblastí.
- Časom môže nastať situácia, že už nie je možné obslúžiť prichádzajúce požiadavky, hoci v súčte je voľnej pamäte dosť.
- Riešenie:
  - **Kompaktifikácia** (kondenzácia) pamäte – obsadené úseky sa presunú vedľa seba a voľné miesto sa zlúči do jedného bloku.
    - Presúva sa celá alokovaná pamäť, je to príliš náročné.
  - Výmena (swap) – jeden proces sa (dočasne) odloží na disk, aby sa mohol zaviesť nový. Odkladá sa celý proces, zaberá súvislý úsek pamäte.
    - Ktorý sa vyberie? Dlho nečinný (zablokovaný). Dlho v pamäti.
- Oba prístupy tiež umožňujú aby sa segment za behu zväčšoval.

Príklad: Externá fragmentácia



Príklad: Kondenzácia

- Systém s *1MB* pamäte robí kondenzáciu jedenkrát za sekundu. Prenos jedného bajtu trvá *0.5* mikrosekundy. Priemerná dĺžka voľného bloku je *0.4* krát veľkosť priemerného prideleného úseku. Aká časť procesorového času sa spotrebuje na kondenzáciu? Ako často treba robiť kondenzáciu, aby sa nespotrebovalo viac ako *10%* procesorového času?

Príklad: Kondenzácia

- Systém s *1MB* pamäte robí kondenzáciu jedenkrát za sekundu. Prenos jedného bajtu trvá *0.5* mikrosekundy. Priemerná dĺžka voľného bloku je *0.4* krát veľkosť priemerného prideleného úseku. Aká časť procesorového času sa spotrebuje na kondenzáciu? Ako často treba robiť kondenzáciu, aby sa nespotrebovalo viac ako *10%* procesorového času?
- Kapacita pamäte – celková $M_C = 1MB$, voľná M_V, obsadená M_O
  - Platí: $M_C = M_V + M_O$, $M_V = 0.4 \cdot M_O \rightarrow M_O = M_C / 1.4 = \lfloor 2^{10} / 1.4 \rfloor = 748982\ B$
- Doba prenosu
  - jeden bajt: $t_1 = 0.5\ \mu s$,
  - celkovo: $t = M_O \cdot t_1 = 748982 \cdot 0.5 \cdot 10^{-6} = 0.375\ s$.
- Perióda kondenzácií $T = 1\ s$, z toho kondenzácia zaberie $t\ /\ T$, čo predstavuje ***37.5 %***.
- $10\% \geq 100\% \cdot t\ /\ T \rightarrow T \geq 100 \cdot 0.375\ /\ 10 = 3.75\ s$
- Aby kondenzácia zabrala maximálne *10%* času, môže sa robiť každých ***3.75 s***.

Stratégie alokácie

Evidencia pridelených a voľných blokov

- Operačný systém musí vedieť, ktoré časti pamäte sú voľné a ktoré obsadené.
- Bitové mapy (tabuľky)
  - Každá časť (alokačná jednotka) má v tabuľke binárny záznam, voľná/obsadená.
  - Čím menšie bloky, tým efektívnejšie sa pamäť využíva (zaplnenie posledného bloku, interná fragmentácia). Zároveň rastie veľkosť tabuľky.
  - Pri novej požiadavke je potrebné prehľadávať tabuľku a nájsť súvislú postupnosť núl, zodpovedajúcu veľkosti požiadavky.
  - Prehľadávanie tabuľky je drahé.
- Zoznamy segmentov (voľný/obsadený, začiatok, veľkosť)
  - Môže byť usporiadaný podľa adries. Pri uvoľňovaní sa ľahko nájdu sesedia a zlúčia sa. Výhodnejší môže byť dvojito zreťazený zoznam.
  - Pri alokácii sa nájde dostatočne veľký segment a rozdelí sa.

Algoritmy alokácie blokov pamäte

- *First fit* (FF)
  - Prehľadáva zoznam a použije prvý voľný segment s dostatočnou veľkosťou.
  - Rýchly, jednoduchý.
- *Next fit* (NF) – alebo tiež *Circular First Fit*, nezačína vždy od začiatku.
- *Best fit* (BF)
  - prehľadáva celý zoznam a použije voľný segment s najbližšou veľkosťou.
  - Vytvára veľa veľmi malých a teda nepoužiteľných voľných blokov.
  - *First fit* vytvára väčšie voľné bloky, čo je výhodnejšie.
- *Worst fit* (WF)
  - prehľadáva vždy celý zoznam, použije voľný segment ktorý vytvorí najväčší voľný blok.
  - Pomalší než BF, ten skončí ak nájde segmemt s presnou veľkosťou.
  - Rozdeľuje veľké bloky, takže veľké úlohy nebude možné umiestniť.

Implementácia a optimalizácia algoritmov

- Pre urýchlenie prehľadávania je možné použiť samostatné zoznamy pre alokované segmenty (procesy) a voľné segmenty pamäte.
- Pri alokácii stačí prehľadávať zoznam voľných.
- Uvoľňovanie je však pomalšie, pretože musia byť modifikované oba zoznamy.
- Zoznam voľných segmentov môže byť usporiadaný podľa veľkosti (nie adresy).
- Je možné tiež použiť samostatné zoznamy pre rôzne často používané veľkosti segmentov (*Quick Fit*).
  - Hľadanie voľného miesta je veľmi rýchle.
  - Podobne ako pri ostatných s usporiadaním podľa veľkosti, uvoľňovanie segmentu je náročné. Aby fungovalo dobre, musí vyhľadať a zlúčiť susedné voľné segmenty.

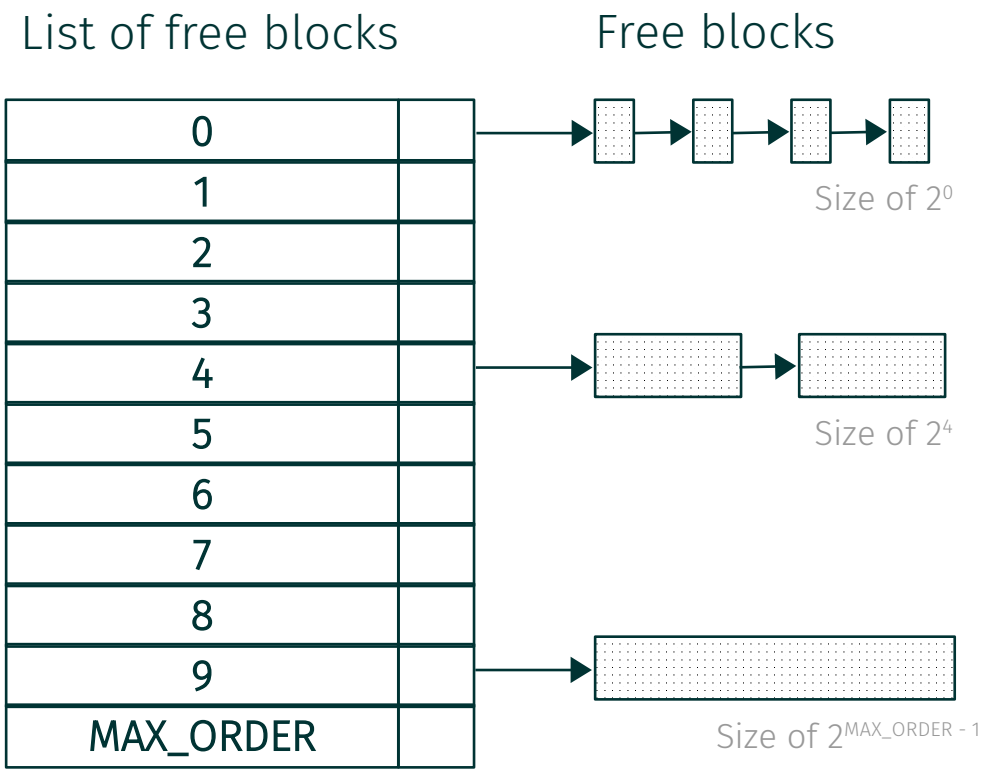
Buddy algoritmus (Knowlton 1965, Knuth 1968)

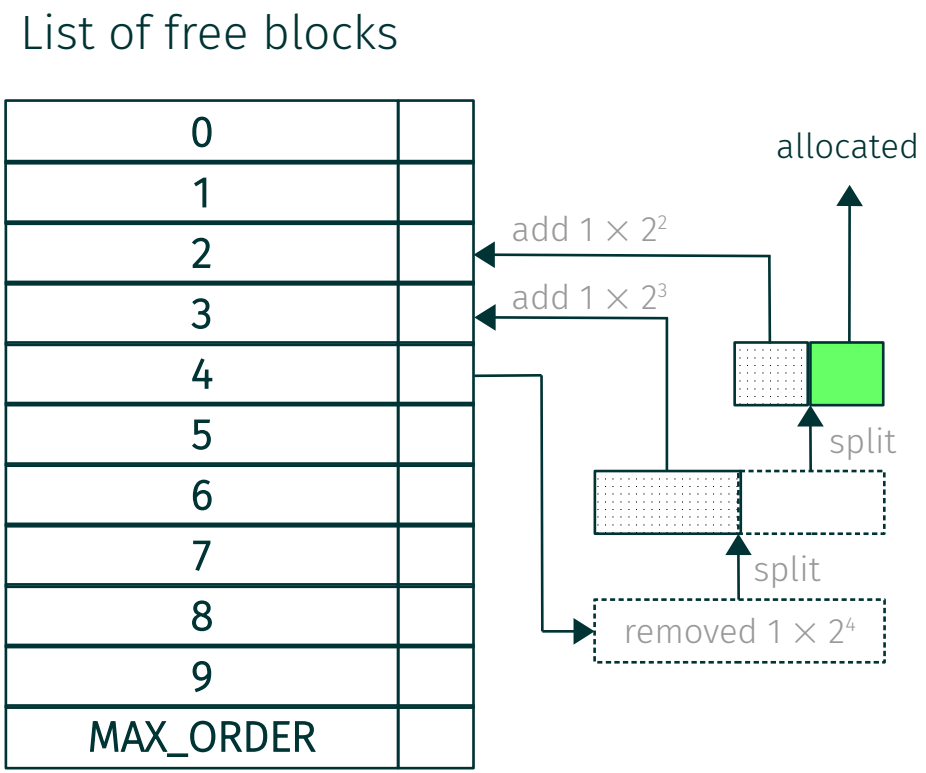
- Udržujú sa zoznamy voľných segmentov s veľkosťou mocnín dvojky.
- Bloky ležiace za sebou tvoria dvojice.
- Pri príchode novej požiadavky sa veľkosť zaokrúhli hore, k najbližšej mocnine *2* a použije sa segment z príslušného zoznamu (ako pri QF).
- Ak je tento zoznam prázdny, nájde sa najbližší vyšší a rozdelí sa na dve časti (ak treba, aj viackrát).
- Pri uvoľnení segmentu sa skontroluje, či je dvojička tiež voľná. Ak áno, zlúčia sa a presunú do vyššieho zoznamu (ak sa dá, aj viackrát).
  - Rozdelovanie a zlučovanie sa robí rekurzívne. V najhoršom prípade však bude veľa zlučovaní nasledovať veľa delení.
- Tvorí základ aj pre alokáciu blokov fyzickej pamäte (stránok) v OS Linux.

Implementácia buddy algoritmu

- Linux využíva bitovú mapu aj zoznam voľných blokov.
  - Jeden bit na každú dvojicu blokov (úspora pamäte).
  - Hodnota sa invertuje pri každej alokácii alebo uvoľnení jedného z nich.
  - Hodnota je 0 ak sú oba bloky voľné, alebo oba obsadené, 1 ak je jeden z dvojice obsadený.
- Pri uvoľňovaní je jasné, že aspoň jeden (práve uvoľnený) je voľný. Ak je hodnota príslušného bitu po invertovaní 0, oba musia byť voľné a môžu sa zlúčiť a preniesť do zoznamu väčších blokov.
  - Adresa oboch blokov v dvojici je zhodná, líši sa len v najnižšom bite ktorý nie je trvale nulový.
- Ak počet voľných blokov klesne pod určitú hranicu, zobudí sa **kswapd** aby uvoľnil miesto odložením niektorých alokovaných častí na disk.

Delenie voľného bloku





Virtuálna pamäť

Obmedzenia správy pamäte

- Pre všetky opísané systémy správy pamäte platí:
  1) Program zaberá vždy súvislý úsek fyzickej pamäte.
  2) Dôsledok: Veľkosť programu je obmedzená dostupnou fyzickou pamäťou.
- Programátor mohol rozdeliť kód do viacerých častí (*overlay*), pričom zabezpečil vždy nahratie tej časti, ktorá sa mala vykonávať.
  - Časť trvale prítomná v pamäti zabezpečovala funkcie zavedenia a nahradenia jednotlivých častí.
  - Je to zložité. Za všetko je zodpovedný programátor.
- Virtuálna pamäť odstraňuje obe obmedzenia a prináša ďalšie výhody.
  - Správa pamäte je realizovaná v OS.

Virtuálna pamäť

- **Virtuálna pamäť**: Proces (programátor) pracuje so súvislým adresovým priestorom, ktorý je operačným systémom mapovaný do fyzickej pamäte.
- Preklad virtuálnych adries na fyzické je pre proces transparentný.
  - Mapovanie môže byť v podstate ľubovoľné.
  - **Relokáciu** vo fyzickej pamäti rieši operačný systém.
  - Proces nemusí byť v súvislom úseku fyzickej pamäti. Umožňuje to eliminovať externú fragmentáciu. Vďaka prekladu adries je možné alokovať celú fyzickú pamäť.
- OS môže pri preklade kontrolovať hranice úsekov. Rieši problém **ochrany** pamäte pridelenej jednotlivým procesom.
- Keďže preklad adries sa robí pri každom prístupe do pamäte, musí byť efektívny.
  - Efektívna implementácia virtuálnej pamäte vyžaduje podporu hardvéru.
  - Preklad adries robí jednotka procesora MMU (*Memory Management Unit*), ktorá generuje fyzické adresy.

Stránkovanie

- Mapovať každú jednu adresu samostatne by bolo príliš náročné.
- Virtuálny adresový priestor je rozdelený na súvislé bloky rovnakej dĺžky (mocniny dvoch) – stránky (*Page*).
  - Typická veľkosť stránky je 4kB.
- Fyzická pamäť je analogicky rozdelená na rovnako veľké bloky – stránkové rámy (*Page Frame*).
- Virtuálna adresa konkrétneho slova obsahuje číslo stránky a posunutie (*offset*) od začiatku stránky.
- Mapovanie priraďuje stránkovým rámom stránky a je realizované tzv. tabuľkou stránok.
  - Indexom do nej je číslo stránky, obsahuje položku (*Page Table Entry*) s číslom stránkového rámu a príznakmi (napr. *present/absent, valid, dirty bit, ...*).
- Každý proces má vlastný virtuálny adresový priestor a teda aj vlastnú tabuľku stránok.

Stránkovanie (nie na požiadanie)

- Stránkovanie nie je to isté ako stránkovanie na požiadanie.
  - *(non-demand) paging* $\neq$ *demand paging*
- Pri stránkovaní (*paging*) musí byť celý vykonávaný proces v pamäti.
- Stránkovanie samo o sebe **rieši externú fragmentáciu** (netreba kompakciu).
  - Celá fyzická pamäť je rozdelená na stránky ktoré môžu byť alokované a teda nie je žiadna nevyužitá voľná pamäť.
  - Mapovanie vytvára súvislý virtuálny adresový priestor, hoci stránky môžu byť rozhádzané v rôznych stránkových rámoch.
- Vzniká **interná fragmentácia**, v priemere polovica stránky na proces.
- Nie je potrebný algoritmus hľadania vhodnej stránky na pridelenie, lebo všetky stránky sú rovnaké. Stačí zobrať prvú voľnú.
- Nie je potrebné spájať voľné miesto po uvoľnení stránky.

Stránkovanie na požiadanie

Stránkovanie na požiadanie (Fotheringham, 1961)

- Virtuálna pamäť umožňuje implementáciu stránkovania na požiadanie (*demand paging*).
- Proces sa môže vykonávať aj keď nie je v pamäti zavedený celý. Stačí ak je rezidentná len časť, ktorá je aktuálne potrebná. Nevyužívané stránky, môžu byť odložené na disk.
- Niektoré výhody:
  - Veľkosť procesu nie je obmedzená fyzickou pamäťou.
  - Umožňuje to zaviesť do pamäti ešte viac procesov, teda zvýšiť úroveň multiprocesingu a tiež využitie procesora.
  - Jednoduchšie pre používateľa, preklad adries je transparentný.
- Niektoré nevýhody:
  - Zložitejší operačný systém.
  - Čas vykonávania procesu sa môže meniť, nie je predvídateľný.
  - OS môže prideliť viac pamäte, než je dostupnej (*overcommitting*).
- V dnešných OS sa využíva predovšetkým tento systém správy pamäte.

Preklad adresy

- Virtuálna adresa obsahuje časť určujúcu stránku a posunutie slova od začiatku stránky.
  - Posunutie (*offset*) je určené spodnými bitmi adresy. Napr. Pri $4kB = 2^{12}$ *B* stránke musí mať adresa pre *offset* vyhradených 12 bitov.
  - Na 32-bitovom systéme by sme mohli mať 32 – 12 = 20 bitov pre *číslo stránky*, čo umožňuje mať 2^{20} stránok. *Stránkových rámov* ovšem môže byť menej.
- Číslo stránky určuje položku v tabuľke stránok.
  - Pokiaľ je nastavený *present bit*, použije sa číslo stránkového rámu na výpočet fyzickej adresy.
  - Číslo stránkového rámu môže byť zhodné s jeho fyzickou adresou.
  - *Offset* stačí pripočítať, nemení sa, keďže stránky a rámy sú rovnako veľké.

Výpadok stránky

- Ak *present bit* v PTE nie je nastavený, nastáva tzv. výpadok stránky (*page fault*).
  - Procesor odovzdá riadenie obsluhe výnimky, teda OS.
- Ak nie je nastavený *valid bit*, adresu nie je možné preložiť, pretože daná virtuálna adresa nezodpovedá alokovanej oblasti.
  - Operačný systém pošle procesu signál SIGSEGV.
- Ak je adresa platná, ale stránka nie je v pamäti, nájde sa voľný stránkový rám.
  - **`kswapd`** sa stará o to, aby vždy nejaké boli. Na základe algoritmu výberu obete zvolí stránku, ktorá bude uvoľnená.
  - Ak je to potrebné, obeť má nastavený *dirty bit*, najskôr sa vybraná stránka zapíše na disk (swap). Aktualizuje sa PTE obete.
- Po alokovaní stránky do voľného stránkového rámu sa aktualizuje PTE a OS reštartuje proces od inštrukcie, ktorá výpadok spôsobila.

Príklad: Výpadok stránky

- Vykonanie jednej inštrukcie trvá jednu mikrosekundu. Ak spôsobí výpadok stránky tak je potrebné vykonať ďalších n inštrukcií pri obsluhe výpadku. Aký je efektívny čas vykonania jednej inštrukcie ak výpadok stránky nastáva každých k inštukcií?

Príklad: Výpadok stránky, riešenie

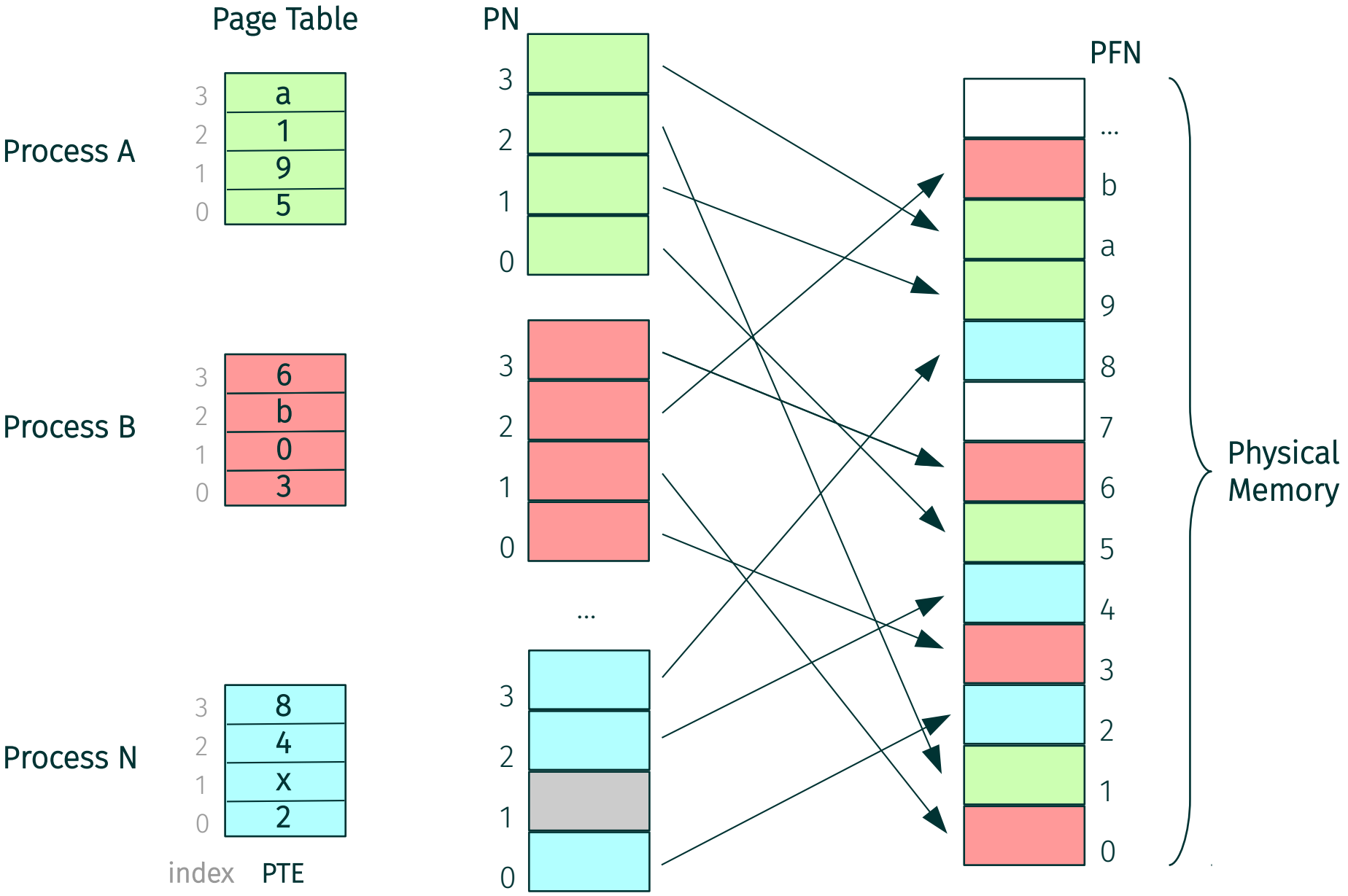
- Vykonanie jednej inštrukcie trvá jednu mikrosekundu. Ak spôsobí výpadok stránky tak je potrebné vykonať ďalších *n* inštrukcií pri obsluhe výpadku. Aký je efektívny čas vykonania jednej inštrukcie ak výpadok stránky nastáva každých *k* inštukcií?
- Vykonanie jednej inštrukcie $t = 1\mu s$.
- Nech celkový počet vykonaných inštrukcií je N.
  - Počet inštrukcií s výpadkom: $N_1 = N / k$
  - Počet inštrukcií bez výpadku: $N_2 = N - N / k$
- Celkový čas vykonávania: $T = t \cdot (N_1 + N_2)$
  - $T = N / k \cdot (1 + n) \cdot t + (N - N / k) \cdot t = N \cdot t \cdot (1 + n / k)$
- Priemerný čas vykonania inštrukcie: $t_E = T / N = t (1 + n / k)$
- Výpadky sú drahé ($n \approx 1000$).

Tabuľka stránok

Tabuľka stránok

- Tabuľka stránok (*PT – Page Table*) sa používa na preklad logickej adresy na fyzickú.
  - Definuje mapovanie stránok na stránkové rámy.
  - Obsahuje položky tabuľky stránok (*PTE – Page Table Entry*), v podstate je to pole PTE.
- Každá PTE zodpovedá jednej stránke a obsahuje číslo stránkového rámu a príznaky.
  - Význam jednotlivých bitov v PTE definuje hardvér (procesor).
- Virtuálna a fyzická adresa nemusia byť rovnako veľké.
- Tabuľky stránok sú dnes spravidla umiestnené v pamäti.
  - Niektoré procesory s menším počtom stránok mali PT v registroch.
  - Napríklad: XDS-940 (Xerox Data Systems), predtým SDS (Scientific Data Systems): 8 stránok po 4096 slov.

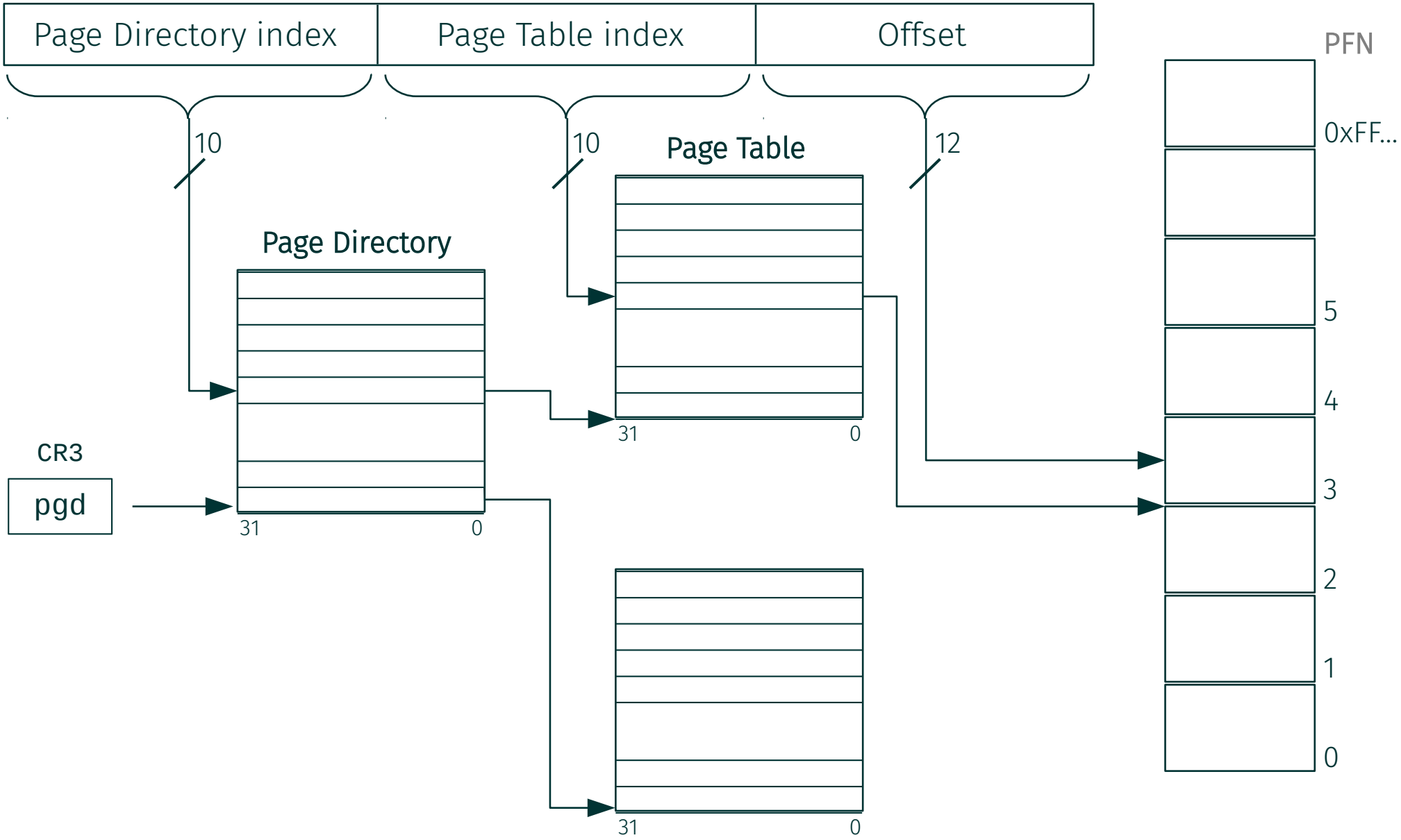
Mapovanie virtuálnej pamäte procesov do fyzickej



Viacúrovňové tabuľky stránok

- Veľkosť jednoúrovňovej tabuľky stránok pri 32-bitovej virtuálnej adrese a 4 kB stránkach:
  - Počet PTE = 2^{20} = 1 048 576
  - Veľkosť PTE = 4B
  - Veľkosť PT = $2^{20} \cdot 4$ B = 2^{22} B = **4 MB**.
- Napríklad pre 100 procesov v systéme by tabuľky zaberali 400 MB.
- Bežný proces pritom nepotrebuje celý 4 GB pamäťový rozsah, čiže väčšina položiek v tabuľke stránok je nevyužitá.
- Tabuľky stránok musia byť umiestnené v súvislých blokoch pamäti (procesor pozná len začiatok tabuľky).
- Riešenie:
  - viacúrovňové tabuľky stránok, tabuľka stránok sa rozdelí na viaceré tabuľky,
  - v pamäti budú len tie, ktoré proces skutočne potrebuje.

Dvojúrovňové tabuľky stránok na 32-bit *x86*



Stránkový adresár

- Stránkový adresár (*PD – Page Directory*) je jeden.
  - Má 2^{10} = 1024 položiek, každá má 4B.
  - Veľkosť PD je 4 kB.
- Každá položka v stránkovom adresári (*PDE – Page Directory Entry*) ukazuje na začiatok jednej tabuľky stránok.
  - PDE a PTE majú podobnú štruktúru, viaceré príznaky sú zhodné.
- Každá položka v tabuľke stránok ukazuje na fyzickú adresu stránkového rámu. Po pripočítaní offsetu vznikne fyzická adresa konkrétneho slova.
- Ak je nastavený bit 7 (*Page Size bit*) znamená to, že PDE neukazuje na tabuľku stránok, ale priamo na stránku.
  - Bity *Page Table index* sa pridajú k *offsetu*. Stránka má teda veľkosť $2^{(10+12)}$ = 4 MB.
  - Umožňuje to zníženie réžie pri alokovaní väčších blokov pamäte.
  - Linux nepoužíva pre stránky procesov tento bit v PDE a teda ani bit 7 v PTE.

Príklad: Dvojúrovňové stránkovanie

- Počítač má 32-bitový adresový priestor a používa dvojúrovňové tabuľky stránok s 9-bitovým indexom do stránkového adresára a 11-bitovým indexom do tabuľky stránok.
  - Aké veľké sú stránky a koľko ich je vo virtuálnom adresovom priestore procesu?
  - Akú časť adresového priestoru je možné adresovať pomocou jednej položky stránkového adresára a aká je s tým spojená réžia?

Príklad: Dvojúrovňové stránkovanie, riešenie

- Počítač má 32-bitový adresový priestor a používa dvojúrovňové tabuľky stránok s 9-bitovým indexom do stránkového adresára a 11-bitovým indexom do tabuľky stránok.
  - Aké veľké sú stránky a koľko ich je vo virtuálnom adresovom priestore procesu?
- $n = 32\ b$, $n_{pd} = 9\ b$, $n_{pt} = 11\ b$: $n_{offset} = 32 - 9 - 11 = 12b$, $S_{page} = 2^{n_{offset}} = 2^{12} =$ **4 kB**.
- Počet stránok: $N_{page} = 2^n / S_{page} = 2^{(n - n_{offset})} = 2^{32-12} = 2^{20} =$ **1 048 576 stránok**.
  - Akú časť adresového priestoru je možné adresovať pomocou jednej položky stránkového adresára a aká je s tým spojená réžia?
- Počet PTE v jednej PT: $N_{pt} = 2^{n_{pt}} = 2^{11} = 2048$
- Jedna PDE adresuje rozsah: $N_{pt} \cdot S_{page} = 2^{11} \cdot 2^{12} = 2^{23} =$ **8 MB**
- Réžia: v pamäti musí byť PD a jedna PT, teda $S_o = S_{pd} + S_{pt}$
- $S_{pd} = 2^{n_{pd}} \cdot 4\ B = 2^9 \cdot 4\ B = 2\ kB$, $S_{pt} = 2^{n_{pt}} \cdot 4\ B = 2^{11} \cdot 4\ B = 8\ kB$, $S_o = S_{pd} + S_{pt} =$ **10 kB**. ■
- Réžia na celý adresový priestor pri jednoúrovňovom stránkovaní by bola $S'_o = S'_{pt} = 4\ MB$.
- Sú to údaje len pre jeden proces. Pri dvojúrovňových tabuľkách stránok je menšia réžia.

Efektívnosť prekladu adries

Efektívnosť prekladu adresy

- Prístup do pamäti je častý. Pre každú vykonávanú inštrukciu aspoň jeden (*instruction fetch*), niektoré pracujú aj s operandami v pamäti.
- Preklad virtuálnych adries na fyzické musí byť efektívny.
  - Preklad robí samostatný hardvér, časť procesora MMU (*Memory Management Unit*).
- Viacúrovňové stránkovanie réžiu znižuje, ale pri dnešných veľkostiach pamäte nie je možné mať všetky potrebné štruktúry v registroch.
- Problém: Pre každý prístup do pamäte sú teraz potrebné **dva prístupy navyše** (PD a PT). Rýchlosť klesá na tretinu.
- Riešenie: Často využívané položky z PT sa uchovávajú v zvláštnej *cache* – TLB (*Translation Lookaside Buffer*) priamo v procesore.

Návrh TLB

- Čo všetko obsahuje?
  - Celú tabuľku stránok, alebo len aktuálne (nedávno) používané položky.
- Ako sa zavádza obsah?
  - Transparentne, alebo softvérovo, privilegovanými inštrukciami (OS).
- Ako sa ruší obsah?
  - Celý obsah naraz, alebo jednotlivé položky.
- Obsah?
  - Len položky jedného procesu, alebo obsah spoločný a položky doplnené značkou adresového priestoru.
- Jednoúrovňové / viacúrovňové ?
- Hardvérová realizácia?

Translation Lookaside Buffer na *x86*

- Asociatívna pamäť v MMU. Obsahuje niekoľko naposledy použitých položiek z tabuľky pre aktuálne bežiaci proces. Kľúčom je PN, hodnotou PTE s hľadaným PFN.
  - Štruktúra: (PN | PTE), číslo stránky slúži na adresovanie asociatívnej pamäte (*tag*).
  - Hľadané PN sa porovnáva so všetkými položkami v TLB naraz.
- Napĺňa ju transparentne procesor, pri prvom prístupe k stránke.
  - V prípade výpadku nájde voľné miesto a nahradí ho potrebnou položkou. OS o výpadku nevie.
- OS musí zabezpečiť, aby v TLB boli len položky pre bežiaci proces.
  - Jednotlivé položky je možné zrušiť inštrukciou `INVLPG`.
    - Operandom je adresa. Inštrukcia zruší záznamy všetkých stránok ktoré ju obsahujú.
  - Platnosť všetkých položiek sa zruší zápisom do registra CR3. Pri zmene kontextu OS zapíše do CR3 hodnotu `mm_struct->pgd` z `task_struct`.
  - Položky (PTE) s nastaveným príznakom G (*global*) zrušené nebudú (ak je to nastavené v CR4). Napríklad stránky jadra alebo stránky zdieľané viacerými procesmi (knižnice).
    - V multiprocesorovom systéme je potrebné zrušiť TLB na všetkých CPU (*TLB shootdown IPI*).

Problémy TLB

- Prepnutie kontextu je drahé aj kvôli réžii spojenej s výpadkom TLB.
  - Výpadok v TLB je pomerne drahý, preto je výhodné spoločné položky zachovať.
- Väčšie stránky umožňujú pokryť viac adries jednou PTE (menej výpadkov).
  - Zväčšuje sa však aj interná fragmentácia.
- Výpadok položky v TLB spôsobí, že bude potrebné prečítať položku z pamäte.
  - Tabuľky stránok sa nachádzajú v logickom adresovom priestore jadra.
  - Pri *HW-managed* TLB musia mať štruktúru ktorú procesor rozozná.
- Následne môže teda dôjsť k ďalšiemu výpadku, ak PTE pre stránku obsahujúcu hľadanú PT nie je v TLB.
- Riešenie: OS si môže udržiavať softvérovú *cache*, ktorá bude v globálnej stránke (teda bude vždy prítomná v TLB).

Translation Lookaside Buffer na *SPARC*

- Obsah TLB spravuje softvér, nastavuje ho operačný systém.
  - Prekladovú mapu môže OS uchovávať v ľubovoľnej štruktúre.
- Výpadok pri hľadaní PTE spôsobí výnimku, ktorú obsluhuje OS.
  - Je to pomalé, preto treba výpadky minimalizovať.
  - Pokiaľ je to možné, položky by mali zostať v TLB aj pri prepnutí kontextu.
  - Niektoré môžu byť označené ako globálne (stránky jadra).
- Štruktúra položky v TLB je doplnená o identifikátor procesu, respektíve adresového priestoru (ASID – *Address Space Identificator*, alebo *Context ID*).
  - Pri hľadaní PFN musí byť zhoda nie len v PN, ale aj *Context ID*.
  - Keď je proces naplánovaný na vykonanie, v TLB môžu byť ešte niektoré jeho položky.

TLB - porovnanie *x86* a *SPARC*

- Pentium–M má štyri druhy TLB (*hardware managed*):
  - 128 položiek pre 4 kB stránky inštrukcií, (*4-way associative*),
  - 2 plne asociatívne položky pre veľké stránky inštrukcií,
  - 128 položiek pre 4 kB stránky s dátami, (*4-way associative*),
  - 8 položiek pre veľké stránky s dátami, (*4-way associative*).
  - Všetky využívajú stratégiu LRU pre uvoľnenie miesta.
- UltraSPARC III má päť druhov TLB (*software managed*):
  - 16 plne asociatívnych položiek pre všetky veľkosti stránok inštrukcií,
  - rovnako pre dáta,
  - 128 položiek pre 8 kB stránky inštrukcií (*2-way associative*),
  - 512 položiek pre stránky s dátami (*2-way associative*), pre dve rôzne veľkosti stránok.
  - Podporuje veľkosti stránok 8 kB, 64 kB, 512 kB, 4 MB.
  - Obsahuje 13 bitový *Context ID*.

Príklad: Účinnosť TLB

- Logický adresový priestor tvorí 1024 stránok. Tabuľky stránok sú v pamäti. Čítanie slova z pamäti trvá 500 ns. Na zníženie tejto réžie má počítač asociatívnu pamäť, ktorá má miesto pre 32 párov (stránka, stránkový rám) a dokáže vyhľadať položku za 100 ns.
  - Aká je úspešnosť vyhľadávania v asociatívnej pamäti (p_{HIT})?
  - Aká je priemerná doba vyhľadania položky (t_{PT})?
  - Aká úspešnosť asociatívnej pamäte je potrebná, aby priemerná doba vyhľadania položky nepresiahla 200 ns?

Príklad: Účinnosť TLB, riešenie

- Logický adresový priestor tvorí 1024 stránok. Tabuľky stránok sú v pamäti. Čítanie slova z pamäti trvá 500 ns. Na zníženie tejto réžie má počítač asociatívnu pamäť, ktorá má miesto pre 32 párov (stránka, stránkový rám) a dokáže vyhľadať položku za 100 ns.
- Aká je úspešnosť vyhľadávania v asociatívnej pamäti (p_{HIT})?
  - $p_{HIT} = N_{TLB} \,/\, N_{page} = 32 \,/\, 1024 = 1 \,/\, 32 =$ ***0.03125***
  - Toto platí za predpokladu rovnomerného rozdelenia adries. V skutočnosti to môže byť lepšie.
- Aká je priemerná doba vyhľadania položky (t_{PT})?
  - $t_{MEM} = 500\ ns,\ \ t_{TLB} = 100\ ns,$
  - $t_{PT} = p_{HIT} \cdot t_{TLB} + (1 - p_{HIT}) \cdot t_{MEM} = 0.03125 \cdot 100\ ns + (1 - 0.03125) \cdot 500\ ns =$ ***487.5 ns***
- Aká úspešnosť asociatívnej pamäte je potrebná, aby priemerná doba vyhľadania položky nepresiahla 200 ns?
  - $200\ ns = t'_{PT} \geq p_{HIT} \cdot t_{TLB} + (1 - p_{HIT}) \cdot t_{MEM}$
  - $p_{HIT} \geq (t'_{PT} - t_{MEM}) \,/\, (t_{TLB} - t_{MEM}) = (200 - 500) \,/\, (100 - 500) =$ ***¾***
  - Táto hodnota je odvodená bez ohľadu na distribúciu adries.

■

64-bitová architektúra

- Ak by proces na 32-bitovej architektúre využíval celý adresový priestor, všetky PTE by typicky zaberali 4 MB.
  - Je to pomerne veľa, ale dá sa to zvládnuť.
- 64-bitové systémy (x86) používajú 48-bitovú logickú a 52-bitovú fyzickú adresu s 4 kB stránkami. PTE má 8 B, z toho 40 bitov PFN.
- Na 64-bitových systémoch by kompletná tabuľka 4 kB stránok zaberala $2^{52} \cdot 8$ B = 2^{55} B = 32 PB, čo je neúnosné.
- Pri dvojúrovňovom stránkovaní by bol stránkový adresár zrejme príliš veľký (index 48 – 10 – 12 = 26 bitov, max. veľkosť 64 MB). Nezmestil by sa do jednej stránky (ani 4 MB).
- Možné riešenia:
  - Hierarchia tabuliek stránok (*Multi-level page tables*).
  - Inverzné tabuľky stránok.
  - Rozptylové tabuľky (*hash tables*).

Multi-level page tables

- Viacúrovňové stránkovanie. Na vyhľadanie PTE sa použije strom tabuliek stránok.
  - Na x86 sa používajú 4 úrovne s 9-bitovým indexom ($4 \cdot 9 + 12 = 48$). Je to dané hardvérom (procesorom).
  - Ak niektorý z adresárov bude ukazovať priamo do fyzickej pamäte (niektoré úrovne PT sa preskočia), stránky môžu mať veľkosť 4 kB, 2 MB, 1 GB.
- Názvy tabuliek: Page Table, Page Directory, Page Directory Pointer, Page Map Level 4 (PML4).
- Každá tabuľka má $2^9 \cdot 8\ B = 512 \cdot 8\ B = 2^{12}\ B = 4\ kB$ (jedna stránka).
  - Potrebné sú len tie tabuľky ktoré ukazujú na alokované časti adresového priestoru.

Inverzné tabuľky stránok

- Na systéme s 48-bitovou logickou adresou môže byť 2^{36} stránok, pri 1 GB fyzickej pamäte (RAM) 2^{18} stránkových rámov.
- Inverzné tabuľky stránok obsahujú jednu položku pre každý stránkový rám.
- Položka obsahuje informáciu o tom, ktorá stránka ktorého procesu (ktorého logického adresového priestoru) je v ráme umiestnená.
- Pamäť potrebná pre inverzné tabuľky je značne menšia.
- Preklad virtuálnej adresy na fyzickú je však náročnejší.
  - Nestačí použiť číslo stránky ako index do tabuľky.
  - V podstate je potrebné pri každom prístupe do pamäte prehľadať celú tabuľku.
- TLB môže preklad výrazne urýchliť. Pri výpadku musí OS nájsť potrebnú položku.
  - OS môže uchovávať zoznam použitých stránok v rozptylovej (*hash*) tabuľke.
  - Pri obsluhe výpadku OS vloží nájdený pár (PN | PFN) do TLB (*SW-managed*).

Uvoľnenie stránkového rámu

Položka tabuľky stránok na *x86*

PFN – číslo stránkového rámu, resp. jeho fyzická adresa, v ktorom je daná stránka umiestnená.

G – *Global*, stránka je využívaná viacerými procesmi, nap. systémová alebo zdieľaná.

D – *Dirty*, do stránky bolo zapísané, jej aktuálny obsah sa nachádza len v pamäti.

A – *Accessed*, stránka bola použitá (referencovaná, čítalo sa z nej, alebo do nej zapisovalo).

C – *Cache Disabled*, obsah stránky nebude umiestňovaný v *cache*.

W – *Write-Through (1), Write-Back (0)*, režim *cache* pamäte pre túto stránku.

U – *User / Supervisor*, stránka môže byť použitá (neprivilegovaným) procesom, inak len jadrom.

R – *Read / Write*, ak je bit nastavený, do stránky je možné aj zapisovať.

P – *Present / Absent*, stránka je prítomná vo fyzickej pamäti.

Príčiny výpadku stránky

- Výnimku výpadku stránky (*page fault*) vygeneruje procesor ak:
  - proces pristupuje k stránke ktorá nemá v PTE nastavený bit **U**,
  - proces zapisuje do stránky ktorá nemá v PTE nastavený bit **R**,
  - proces pristupuje k stránke ktorá nemá v PTE nastavený bit **P**.
- Pred obsluhou výnimky procesor uloží:
  - na zásobník chybový kód ktorého spodné tri bity sú URP,
  - do CR2 adresu ktorá výnimku spôsobila.
- Rutina obsluhy výnimky podľa týchto informácií rozhodne, čo urobiť.
  - Napríklad ak proces robí prístup v rozpore s nastavením bitov U a R, ide o porušenie ochrany pamäte (napr. pokus o prístup k pamäti jadra) a OS pošle procesu signál SIGSEGV.

Obsluha výpadku stránky

- Ak bit P nie je nastavený, môže nastať viacero prípadov.
  - Stránka nebola nikdy alokovaná, proces dostane SIGSEGV.
  - Nájde sa voľný stránkový rám (ak je to potrebné, uvoľní sa) a upraví sa PTE.
    - Ak je to nová stránka (prvý prístup), koniec.
    - Ak je obsah na disku, nahrá sa do pamäte a koniec.
- Ak bit P nie je nastavený, procesor urobí výnimku bez ohľadu na adresu v PTE.
  - 20 bitov v PTE je v tomto prípade možné použiť priamo ako odkaz na odkladacie médium (*swap*) a číslo bloku v ňom.

Informácie o využití stránok

- Bity **D** (*Dirty, written*) a **A** (*Accessed, refrenced*) v PTE nastavuje procesor.
- Ich nulovanie musí ale zabezpečiť operačný systém.
- Tieto bity môžu byť použité pri hľadaní obete na uvoľnenie stránkového rámu.
- Lepšie môže byť vyhodiť menej často používanú stránku.
  - OS môže pravidelne nulovať *Accessed* bity. Stránky ktoré majú tento bit nastavený boli použité nedávno.
- Výhodnejšie môže byť vyhodiť stránku, ktorá nebola po zavedení do pamäte modifikovaná (napr. text).
  - Modifikovanú stránku je potrebné pred uvoľnením jej stránkového rámu najskôr zapísať (na *swap*, alebo do súboru), čo zvyšuje čas obsluhy.

Algoritmy výberu obete

Algoritmy hľadania obete

- Problém výberu obete pri uvoľňovaní bloku pamäte sa vyskytuje aj v iných úrovniach pamäťového systému, ako napríklad *cache* medzi CPU a hlavnou pamäťou.
- Vybrať náhodnú stránku
  - Jednoduchá implementácia, neuvažuje žiadne informácie o využití stránok. Môže odstrániť stránku, ktorá bude onedlho opäť potrebná.
- Optimálny algoritmus
  - Za obeť vyberie vždy tú stránku, ktorá bude potrebná za nadlhší čas, teda najďalej v budúcnosti.
  - Vo všeobecnosti ho nie je možné implementovať.
  - Pokiaľ sa nejaký program vykonáva vždy rovnako, je možné postupnosť prístupov k stránkam zaznamenať a použiť pri ďalších spusteniach.
  - Optimálny algoritmus dáva najlepšie výsledky a preto sa používa na porovnanie a vyhodnotenie rôznych iných algoritmov.

Not Recently Used (NRU)

- Využíva bity R (Referenced) a D (Dirty, Modified).
- Bit R, nastavený CPU pri každom prístupe k stránke, OS pravidelne nuluje (pri každom prerušení od časovača, rádovo každých 10 ms).
  - To umožňuje rozlíšiť, ktoré stránky boli použité nedávno a ktoré dávnejšie (pred posledným prerušením).
  - Bit D sa nenuluje. Nie je možné ho vynulovať, pokiaľ sa obsah najskôr neuloží.
- Všetky stránky sa rozdelia do štyroch tried, podľa hodnoty bitov R a D.
  - Trieda 0: stránky nepoužité, nezmenené,
  - Trieda 1: stránky nepoužité, zmenené,
  - Trieda 2: stránky použité, nezmenené,
  - Trieda 3: stránky použité, zmenené.
- Za obeť sa vyberie náhodne, spomedzi stránok najnižšej neprázdnej triedy.
- Pomerne jednoduchá implementácia. Stačia informácie z PTE.

First-In, First-Out (FIFO)

- Operačný systém udržiava zoznam všetkých stránok prítomných v pamäti, usporiadaný podľa času, kedy bola stránka zavedená do pamäti.
- Pomerne jednoduchá implementácia.
- Pri výpadku stránky sa zo zoznamu odstráni stránka zo začiatku zoznamu (obeť) a nová sa pridá na koniec.
- Pravidelne však môže odstrániť aj stránky, ktoré sú veľmi často využívané.
- Prakticky sa príliš nepoužíva.

Second chance

- Algoritmus využíva rovnaký zoznam ako FIFO.
- Pokiaľ má najstaršia stránka vynulovaný R bit, bude označená ako obeť.
- Ak má však stránka na začiatku zoznamu R bit nastavený, tento sa vynuluje a stránka sa zaradí na koniec zoznamu, ako by bola nová (práve zavedená).
  - Stránka dostane druhú šancu.
- Ide o modifikáciu FIFO, ktorá však uprednostňuje pri výbere obete stránky nepoužívané (v poslednej dobe).
- Ak všetky stránky v pamäti boli od posledného prerušenia použité, ide o FIFO.
- Réžia spojená s presunmi stránok v zozname je pomerne veľká.

Clock algorithm

- Princíp je zhodný s algoritmom *Second chance*.
- Stránky sú organizované v kruhovom zozname, ale nepresúvajú sa.
- Ukazovateľ (*pointer*) označuje najstaršiu stránku.
- Pokiaľ má R bit nulový, vyberie sa ako obeť. Nová stránku sa zaradí na toto miesto a ukazovateľ sa posunie na ďalšiu.
- Ak je R bit nastavený, vynuluje sa, a ukazovateľ sa posunie na ďalšiu stránku.
- Efektívnejšia implementácia ako SC.
- Používané stránky zostávajú v pamäti.

Least Recently Used (LRU)

- Za obeť sa vyberie najdávnejšie použitá, resp. *najdlhšie nepoužitá*, stránka.
- Algoritmus je založený na predpoklade, že to tak zostane aj v budúcnosti.
- Je to **dobrá aproximácia optimálneho** algoritmu.
- Implementácia je však náročnejšia (v pôvodnej podobe prakticky neúnosná).
- Zoznam uchovávajúci poradie prístupov k stránkam by sa musel aktualizovať po každom prístupe do pamäti, teda pri každej inštrukcii, čo nie je únosné.
- Musel by to robiť hardvér, OS sa nedostane k slovu po každej inštrukcii.
  - Napríklad: hardvérové počítadlo prístupov pre každú stránku, ako súčasť PTE. Obeťou je stránka s najnižšou hodnotou počítadla;
  - Alebo matica, v ktorej sa pri prístupe k stránkovému rámu n nastavia najprv jednotky do n-tého riadku a potom nuly do n-tého stĺpca. Obeťou je stránka zodpovedajúca riadku s najnižšou hodnotou.

Not Frequently Used (NFU)

- Ide o softvérovú implementáciu LRU, respektíve aproximácie LRU.
- Keďže hardvér neposkytuje počítadlo prístupov pre stránky, môže ho implementovať OS.
- Pri každom prerušení sa k počítadlu prístupov pre stránku pripočíta hodnota jej R bitu.
  - Aktualizácia sa teda nerobí pri každom prístupe, ide len o aproximáciu LRU.
- Za obeť sa vyberie stránka s najnižšou hodnotou počítadla.
- Problémom je, že algoritmus nezabúda (je málo adaptívny).
  - Ak niektorá stránka získa vysokú hodnotu počítadla, je uprednostňovaná aj keď už aktuálne nebude využívaná.

Algoritmus starnutia (Aging)

- Modifikácia, a vylepšenie, NFU.
- Hodnota počítadla sa najskôr posunie o jeden bit doprava a potom sa R bit pripočíta na najvyššiu pozíciu.
- Za obeť sa vyberie stránka s najnižšou hodnotou počítadla.
- Ide o dobrú aproximáciu LRU.
- Problémy:
  - Počítadlá sa aktualizujú s periódou časovača. Pri zhode v bitoch z posledného tiku sa rozhoduje na základe bitov z predchádzajúcich. Aktuálne poradie však už môže byť iné.
  - Počítadlo má konečnú šírku. Pri zhode sa vyberá náhodne (čo je rozdiel voči LRU a nemusí to byť dobre).

Algoritmy – praktické zhrnutie

Zhrnutie

- Optimálny algoritmus
  - Založený na znalosti budúceho správania. Ideálny, v princípe nerealizovateľný. Dobrý na porovnávanie.
- Least Recently Used (LRU)
  - **Dobrá aproximácia optimálneho.**
  - Vyžaduje aktualizáciu zoznamov pri každom prístupe do pamäte. Prakticky nerealizovateľný (v HW aj SW).
- First-In, First-Out (FIFO)
  - Vyhodí najstaršiu stránku bez ohľadu na to, či sa používa. Prakticky nepoužiteľný.
- Clock
  - Vychádza z FIFO, berie do úvahy aj R bit (Second Chance). Namiesto preusporiadania zoznamu používa ukazovateľ (ručička hodín).
  - **Dobrá aproximácia LRU.**

Zoznamy aktívnych a neaktívnych stránok

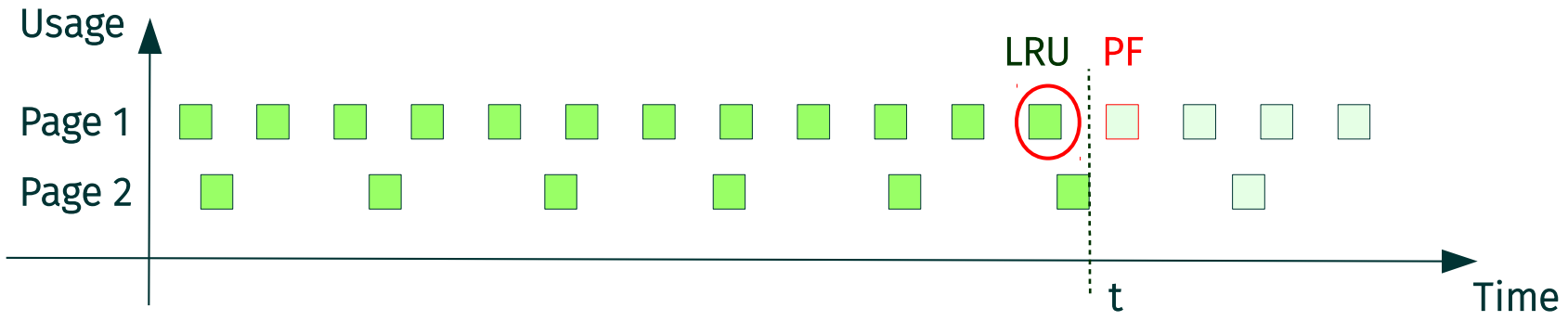
- Pri praktickej realizácii by sledovanie zmien vo využívaní všetkých stránok predstavovali vysokú réžiu.
- OS udržuje dva zoznamy stránok: aktívne a neaktívne (*active/inactive list*).
  - Na prvom zozname sú stránky ktoré sa používajú (pracovné množiny procesov), na druhom stránky, ktoré sa už asi používať nebudú (kandidáti na vyhodenie).
  - Pravidlá pre presun stránok medzi zoznamami sa líšia podľa typu mapovania stránky (môžu mať vlastné zoznamy).
- Keď sa stránka presunie do zoznamu neaktívnych vynuluje sa *Present bit* v PTE. Stránka však zatiaľ zostáva v pamäti.
- Pokiaľ sa vyskytne prístup k stránke zo zoznamu neaktívnych, nastane (*soft*) *page fault*. Obslúži sa rýchlo, lebo stránka je v pamäti a presunie sa do zoznamu aktívnych.
- Ak je zoznam aktívnych stránok väčší než zoznam neaktívnych, niektoré stránky sa presunú.

Pravidlá a vlastnosti algoritmu

- Stránky mapované na súbory sa častokrát použijú len raz.
  - Po zavedení do pamäti sa zaradia do zoznamu neaktívnych (prečo?).
  - Pokiaľ sa použijú opakovane, presunú sa do zoznamu aktívnych.
- Stránky anonymnej pamäte sa zväčša používajú často.
  - Po alokovaní sa zaradia do zoznamu aktívnych (na začiatok).
  - Stránka na konci zoznamu aktívnych sa presunie do neaktívnych.
- Zoznam neaktívnych stránok používa starnutie (*aging*).
- Stránka na konci zoznamu neaktívnych sa vyberie ako obeť a uvoľní miesto.
- Pokiaľ stránky môžu zostať v zozname neaktívnych dosť dlho na to, aby mali šancu vrátiť sa medzi aktívne, ide o dobrú aproximáciu LRU.
  - Algoritmus má podobné správanie ako LRU, prakticky je efektívny.

Predpoklady LRU

- Ani LRU nedáva dobré výsledky pre všetky vzory prístupov.
  - Problémom je napríklad jednorazový alebo cyklický prístup k stránkam.
- Môže sa stať, že stránka ktorá bola použitá dávnejšie, bude čoskoro použitá znova, hoci nedávno použitá stránka bude dlhšie nepotrebná.
  - LRU by ju však označil za obeť a následne by nastal výpadok.
  - LRU neberie do úvahy frekvenciu prístupov k stránkam.



Algoritmus Clock-Pro (Jiang, 2005)

- Metriky lokálnosti prístupov:
  - *R(p) – Recency*, koľko prístupov k iným stránkam sa urobilo od posledného prístupu k stránke *p*.
  - *RD(p) – Reuse distance*, (alebo aj *Inter-Reference Recency*) počet iných stránok ku ktorým sa pristupovalo medzi dvoma prístupmi k stránke *p* (medzi posledným a predposledným prístupom k *p*).
- Algoritmus LRU vyberie ako obeť stránku v pamäti s najväčším *R(p)*.
- Clock-Pro vyberie ako obeť stránku na základe *RD(p)* (s najväčšou hodnotou).
  - Na určenie RD uchováva aj informácie o stránkach, ktoré už nie sú v pamäti.
  - Podobne ako pri pôvodnom Clock algoritme, pri prístupe k stránke v pamäti nie sú potrebné žiadne akcie OS, len nastavenie R bitu hardvérom.
  - V zozname neaktívnych udržuje stránky ku ktorým sa pristupuje menej často, než k tým na zozname aktívnych.
- Napríklad Linux používa algoritmus založený na Clock-Pro.

Page cache

- Operačný systém časť stránkových rámov vyhradí pre *cache*.
  - V OS Linux typicky 1 až 15% z celkovej veľkosti pamäte.
- Stránky prečítané z disku zostávajú v pamäti aj keď ich už žiadny proces nepotrebuje.
  - Nie sú namapované vo virtuálnom adresovom priestore žiadneho procesu.
- Keďže tieto stránky žiadny proces práve nepotrebuje, pri nedostatku pamäte môžu byť bez väčšieho dopadu uvoľnené.
- V prípade, že stránky budú opäť potrebné, budú už v pamäti.

Princípy pamäťovej lokality

Hierarchia pamäťového systému

- Počítačové systémy obsahujú viacero typov pamäte, založených na rôznych technológiách, s rôznym typom prepojení.
  - Vyššia rýchlosť → zložitejšie/drahšie → menšia kapacita,
  - a naopak.
- Registre (CPU), ~1kB, < 1ns, SRAM
- Cache (CPU), L1, L2, L3, D/I, ~10MB, < 10ns, SRAM
- Hlavná pamäť (RAM), ~100GB, < 100ns, DRAM
- SSD, ~100MB, < 1ms, Flash
- HDD, ~1TB, < 10ms
- Sieťové úložiská, >> 1TB, > 10-100 ms

Vlastnosti pamäťovej hierarchie

- Inkluzívnosť
  - V úrovni pamäte bližšie k CPU je vždy podmnožina údajov zo vzdialenejšej úrovne.
  - Počas spracovania sa podmnožina dát presúva medzi úrovňami smerom k CPU.
  - Hodnota registra môže byť umiestnená v bloku *cache*, ten je v stránke, tá je v diskovom bloku, ten je súčasťou súboru, resp. disku, ...
- Koherencia
  - Inštancie rovnakých dát v rôznych úrovniach musia byť konzistentné.
  - Zmeny v dátach sa musia šíriť aj do úrovní vzdialenejších od CPU (*cache writethrough* – okamžite, *writeback* – keď je to potrebné).
- Lokalita
  - Prístup do pamäte nie je distribuovaný rovnomerne, ale vykazuje isté vzory.

Princíp lokality

- Adresy generované procesmi počas vykonávania nie sú distribuované rovnomerne. Pristupované adresy vykazuje isté vzory.
  - Sekvenčná, priestorová, časová lokalita.
- Proces vždy pomerne dlho využíva istý malý interval adries, ktorý sa mení pomerne pomaly.
- Tento interval by sa mal nachádzať čo najbližšie k CPU.
- Efektívna doba prístupu k dátam závisí od toho v ktorej úrovni sa nachádzajú, respektíve od počtu prenosov medzi rôznymi úrovňami.

Sekvenčná lokalita

- Ak procesor v čase *n* pristupuje k adrese *a*, tak s vysokou pravdepodobnosťou bude v čase *n+1* pristupovať k adrese *a+1*.
- Sekvencia je najčastejšou programovou konštrukciou.
  - Čítanie inštrukcií z pamäte ide postupne, s výnimkou vetvenia, resp. skokov.
  - Register PC (IP) sa spravidla inkrementuje pri vykonaní každej inštrukcie.
  - Prehľadávanie polí môže viesť k tomuto vzoru aj v dátach.
- Oplatí sa preniesť do vyššej úrovne viac po sebe idúcich blokov naraz (*lookahead*).
  - Nasledujúce adresy pravdepodobne bude čoskoro treba tiež.

Priestorová lokalita

- Ak procesor v čase *n* pristupuje k adrese *a*, tak s vysokou pravdepodobnosťou bude v čase *n+1* pristupovať k adrese blízkej *a* (*a*±*m*).
- Iterácia udržuje program v istých medziach. Po istej dobe sa vráti na rovnakú adresu.
  - Vytvára priestorovú lokalitu prístupov k textu.
- Položky dátových štruktúr sú umiestnené blízko pri sebe.
  - Spracovanie štruktúr môže vytvárať priestorovú lokalitu v dátach.
- Do vyššej úrovne hierarchie sa oplatí prenášať väčšie bloky (nie jednotlivé hodnoty), lebo blízke adresy budú asi tiež potrebné.

Časová lokalita

- Ak procesor v čase *n* pristupuje k adrese *a*, tak s vysokou pravdepodobnosťou bude k adrese *a* znovu pristupovať v blízkom čase (*n+t*).
- Iterácia opakuje prístup k inštrukciám aj dátam.
- Lokálne premenné a argumenty.
  - Funkcia počas svojho vykonávania k nim bude pristupovať opakovane.
- Údaje prenesené do vyššej úrovne by tam mali zotrvať čím dlhšie, lebo väčšinou ich procesor potrebuje opakovane.

Efektívnosť pamäťovej hierarchie

- Procesy môžu bežať rýchlo aj keď je fyzickej pamäte menej než celkový súčet pamäte alokovanej procesmi.
  - Ak používajú vždy len istú malú časť dát ktorú majú alokovanú.
  - Nevracajú sa často k dlho nepoužívanej dátam odloženým na disk.
- Priemerné procesy sa tak skutočne správajú, teda podľa princípov pamäťovej lokality.
- Bežné procesy väčšinou opakovane využívajú len malú časť adresového priestoru a zmeny sú relatívne pomalé.
  - Napríklad po ukončení cyklu, pri volaní alebo ukončení funkcie, po vetvení a pod.
  - Sekvenčné prehľadávanie veľkého poľa alebo náhodné a rovnako pravdepodobné prístupy ku všetkým adresám, sa prakticky nevyskytujú.
- **Predpoklad algoritmu LRU**, že blok dát ku ktorému proces pristupoval nedávno bude v najbližšom čase potrebovať opäť, je **v súlade s princípmi priestorovej a časovej lokality**.
  - Využíva sa na rôznych úrovniach hierarchie (*cache*: CPU ↔ pamäť, stránkovanie: pamäť ↔ disk).

Návrh systému stránkovania

Pracovná množina stránok (Denning, 1968)

- Stránky, ktoré proces práve využíva, tvoria pracovnú množinu stránok (*working set*).
  - Pracovná množina sa počas vykonávania mení, nie však rýchlo.
  - Pri systéme stránkovania na žiadosť proces po štarte nemá žiadnu stránku v pamäti.
- Pokiaľ sú stránky z pracovnej množiny v pamäti, proces sa vykonáva bez väčšieho počtu výpadkov (ideálne s minimálnym počtom).
  - A to aj keď nie je v pamäti celý. Toto je výhoda stránkovania na žiadosť.
- Ak proces nemôže mať celú pracovnú množinu v pamäti, budú počas vykonávania výpadky vznikať neustále.
  - Nech systém používa akýkoľvek algoritmus výberu obete, pravidelne sa vyhodí stránka ktorú onedlho bude nutné znovu zaviesť.
  - Táto situácia, neustále odkladanie a nahrávanie stránok, sa nazýva *trashing*.
  - Vyťažuje disk aj procesor.

Veľkosť pracovnej množiny

- Nech *w(k, t)* je počet (rôznych) stránok ku ktorým proces urobil posledných *k* prístupov v čase *t*.
- S rastúcim *k* rastie aj veľkosť *w(k, t)*, pretože s väčším počtom prístupov v minulosti sa počet použitých stránok môže len zväčšiť.
  - Ide teda o monotónne rastúcu funkciu (v *k*).
  - Pre fixné *k* môže veľkosť pracovnej množiny v čase rásť, aj klesať (pomaly).
- Zároveň je to funkcia zhora ohraničená, keďže adresový priestor a teda aj maximálny počet stránok, sú konečné.
  - Limita *w(k, t)* s rastúcim k sa rovná celkovému počtu všetkých stránok, ktoré proces počas vykonávania potreboval.
- Veľkosť pracovnej množiny vzhľadom na *k* rastie najprv rýchlo, potom už len pomaly. Pre veľký rozsah *k* je teda pracovná množina podobne veľká.
  - Veľkosť pracovnej množiny teda od *k* príliš nezávisí a s *t* sa mení pomaly.

Predstránkovanie (*prepaging*)

- Keďže pracovná množina stránok sa pre proces mení len pomaly, v každom čase je možné odhadnúť, ktoré stránky by mali byť v pamäti, aby proces bežal bez výpadkov.
- Ak bol proces odsunutý z pamäte na záložné médium (*swap*), po opätovnom naplánovaní na vykonávanie nebude jeho pracovná množina v pamäti.
- Systém stránkovania na žiadosť postupne zabezpečí zavedenie potrebných stránok, je to však neefektívne.
  - V tomto prípade určite nastane séria výpadkov.
- Ak by operačný systém vedel ktoré stránky patria do pracovnej množiny procesu, mohol by ich zaviesť vopred – predstránkovanie (*prepaging*).
- Napríklad ak sa používa systém stranutia (*aging*), stránky s nastaveným niektorým z vyšších bitov počítadla pravdepodobne patria do pracovnej množiny.
- *Clock* algoritmus vyhodí najstaršiu stránku s nenastaveným R bitom. Pokiaľ však OS vie, že patrí do pracovnej množiny, môže vybrať inú – algoritmus *wsclock*.

Lokálny vs. globálny prístup

- Má sa obeť vyberať len medzi stránkami procesu ktorý potrebuje pamäť, alebo medzi stránkami všetkých procesov?
- Lokálny prístup:
  - Zodpovedá situácii, že proces má pridelený istý počet stránkových rámov a obeť sa vyberá spomedzi nich.
  - Ak veľkosť pracovnej množiny rastie (nad počet pridelených rámov), môže nastať *trashing*, hoci pamäť je ešte voľná.
  - Ak je pracovná množina menšia, pamäť nie je využitá efektívne.
- Globálny prístup:
  - Je efektívnejší, ak sa veľkosť pracovnej množiny mení.
  - OS musí vedieť aká je veľkosť pracovnej množiny (e.g. *aging*), aby mohol prideliť procesu stránkové rámy.

Page Fault Frequency (PFF)

- Pri globálnom prístupe sa počet pridelených stránkových rámov určuje dynamicky.
- Počet výpadkov stránok spôsobených procesom za nejaký čas môže byť využitý pri stanovení počtu pridelených stránkových rámov.
  - Meria sa pomerne jednoducho.
- Predpoklad: S rastúcim počtom pridelených stránkových rámov klesá počet výpadkov.
  - Pre väčšinu algoritmov výberu obete to platí.
- Algoritmus sleduje frekvenciu výpadkov a pridávaním alebo odoberaním stránkových rámov sa ju snaží udržať v prijateľnom intervale.
  - Algoritmus má dva parametre, hranice intervalu pre zmenu počtu rámov.
- Dnes sa využíva najmä globálny prístup.
- Procesu je možné nastaviť limit pre počet pridelených stránkových rámov.

Príklad: Počet stránkových rámov

- Počet inštrukcií vykonaných medzi dvoma výpadkami stránok je priamo úmerný počtu stránkových rámov pridelených procesu.
- Vykonanie inštrukcie trvá bežbe $1\mu s$ a s výpadkom stránky $2001\mu s$. Program trval $60s$ a mal 15000 výpadkov. Ako dlho by trvalo vykonanie programu s dvojnásobným počtom stránkových rámov?

Faktory ovplyvňujúce voľbu veľkosti stránky

- Program sa spravidla skladá z viacerých segmentov (text, data, stack, ...).
- Veľkosť segmentu nebude vždy celočíselným násobkom veľkosti stránky.
- V dôsledku internej fragmentácie bude v priemere polovica stránky na každý segment nevyužitá. Z tohto pohľadu sú výhodnejšie menšie stránky.
- Proces väčšinou vykonáva dlhšiu dobu kód len z malého rozsahu adries. Ak by stránky boli veľké, pamäť by zbytočne zaberal kód, ktorý sa väčšinou nevykonáva. Podobne to platí aj pre dáta.
- Na druhej strane menšie stránky vyžadujú väčší celkový počet stránok ktoré proces potrebuje a teda aj väčší priestor v tabuľkách stránok.
- Pri prepnutí kontextu môže zavedenie potrebných PTE do TLB trvať dlhšie.
- Pri presune stránok medzi pamäťou a diskom je kvôli latencii prístupu výhodnejšie presúvať menej veľkých blokov než veľa malých.

Optimálna veľkosť stránky

- Veľkosť tabuliek stránok je nepriamo úmerný veľkosti stránky.
- Nech veľkosť priemerného procesu je s, veľkosť PTE e a veľkosť stránky p.
  - Počet stránok na proces s/p a veľkosť PT bude $s \cdot e/p$.
  - Interná fragmentácia pri n segmentoch bude $n \cdot p/2$.
  - Celkový odhad réžie na tabuľku stránok a internú fragmentáciu:
    - $O(p) = s \cdot e/p + n \cdot p/2$
  - Prvý člen je veľký pre malé stránky, druhý člen pre veľké stránky.
  - Extrém (minimum) dosahuje réžia pre veľkosť stránky:
    - $dO(p)/dp = 0 \rightarrow p_{opt} = \sqrt{2 \cdot s \cdot e/n}$

Segmentácia

Segmenty

- Pri stránkovaní je logický adresový priestor procesu lineárny. Všetky adresy sú usporiadané sekvenčne, s jedným začiatkom.
- Program sa však prirodzene skladá z viacerých súvislých blokov so samostatným významom – zo segmentov (text, data, bss, heap, stack, ...).
  - Ich veľkosť sa môže meniť (rásť) nezávisle od iných segmentov.
  - Každý má vlastné hranice a spôsob prístupu (rwx).
  - Dáta – jedno pole – jeden segment – prirodzená kontrola hraníc.
  - Zásobník – môže rásť bez toho, aby hrozilo prekrytie s inými časťami.
  - Obsah jednotlivých segmentov sa môže kompilovať samostatne.
  - Segment môže byť používaný viacerými procesmi (zdieľané knižnice).
- Segmentácia – rozdelenie logického adresového priestoru na samostatné segmenty.

Segmentácia

- Logická adresa sa skladá z čísla segmentu a ofsetu (od začiatku segmentu).
  - Každý segment predstavuje samostatný adresový priestor. Fyzicky sa neprekrývajú.
  - Stránkovanie je pre programátora transparentné, segmentácia nie je.
  - Ochrana a zdieľanie sú pri segmentácii jednoduchšie. Robí to hardvér (MMU).
- Fyzická adresa sa získa súčtom bázovej adresy segmentu vo fyzickej pamäti a ofsetu.
  - Na preklad sa používa tabuľka segmentov. Okrem bázovej adresy obsahuje tiež limit (veľkosť) a typ prístupu (rwx).
- Stránky majú pevnú veľkosť, segmenty premenlivú (aj počas vykonávania).
  - Vedie to k externej fragmentácii (potreba kompakcie).
- Výhodnejšia je kombinácia segmentácie so stránkovaním (všeobecnejší prístup).
  - Každý segment je rozdelený na stránky. Architektúra Intel Pentium to umožňuje.
  - Jediný operačný systém ktorý to využíval bol IBM (MS) OS/2 (1987 – 2001).
  - Ostatné OS využívajú len jeden (fyzický) segment (do neho mapujú programové segmenty).

Virtuálny pamäťový priestor

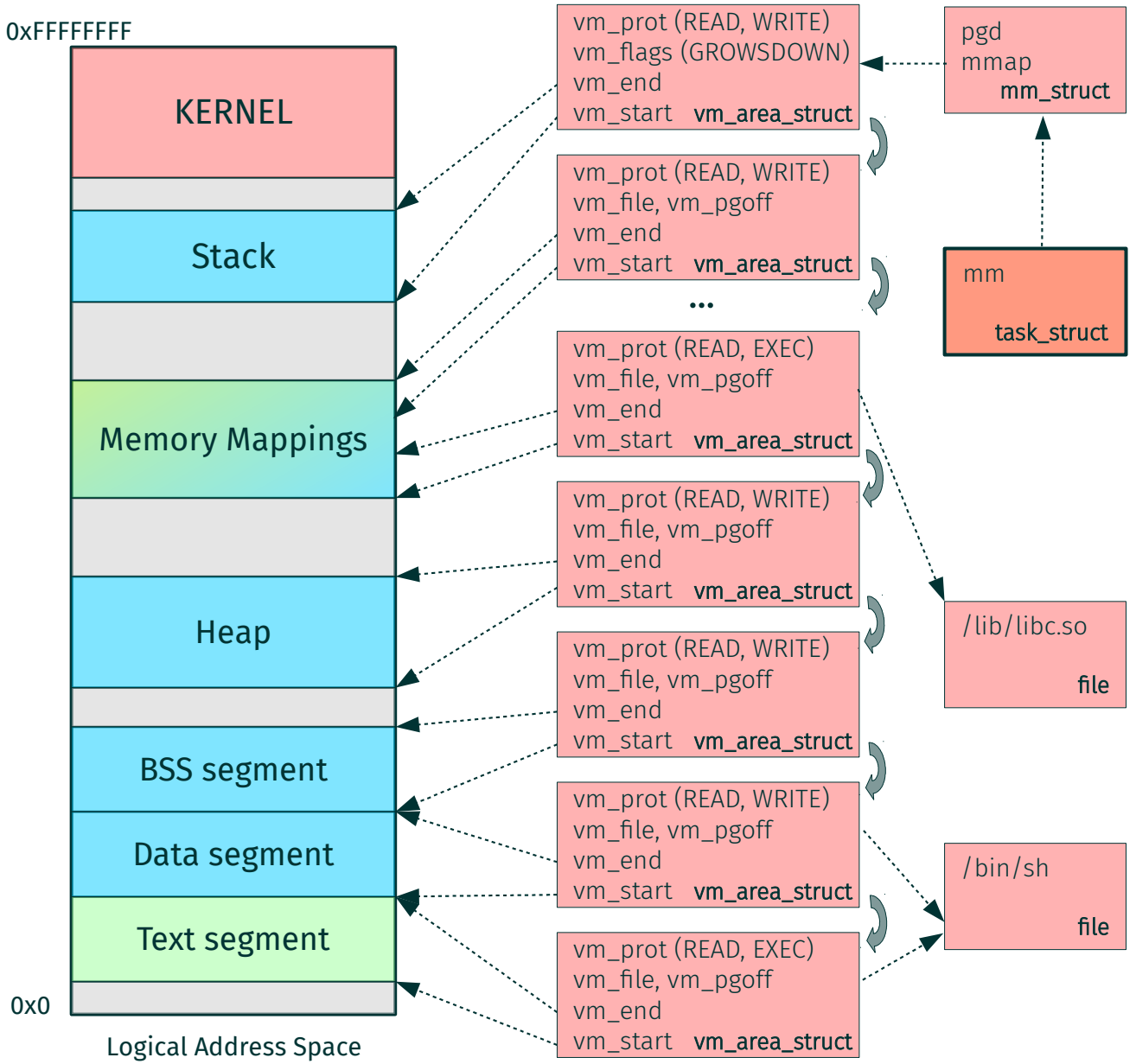
Oblasti virtuálnej pamäte

- Virtuálny adresový priestor je podelený na súvislé oblasti VMA (*Virtual Memory Area*), pričom každá je opísaná v štruktúre typu **`vm_area_struct`**.
  - Obsahuje začiatok (**`vm_start`**), koniec (**`vm_end`**), typ prístupu (a mapovanie na bity v PTE, **`vm_page_prot`**).
  - Ak je rozsah adries namapovaný na obsah súboru, tak **`vm_file`** a **`vm_pgoff`**.
  - Mapovanie bez súboru sa nazýva anonymné, stránky sa ukladajú na *swap*.
  - Obsahuje ukazovatele na operácie s VMA (open, close, mprotect, fault, map_pages, ...).
- Každý proces má vo svojej **`task_struct`** položku **`mm`** (typu **`mm_struct`**), ktorá odkazuje na zoznam VMA (**`mmap`**, **`mm_rb`**) a tabuľky stránok **`pgd`**.
- Každý segment programu (*text*, *data*, *stack*, ...) má vlastnú VMA.
  - OS však nevie ktorá VMA obsahuje ktorý segment, pre OS sú rovnaké.
  - Zoznam dostupný v **`/proc/`*`pid`*`/maps`**.
- Nealokované (nenamapované) virtuálne adresy nespadajú do rozsahu žiadnej VMA. Použité adresy áno.

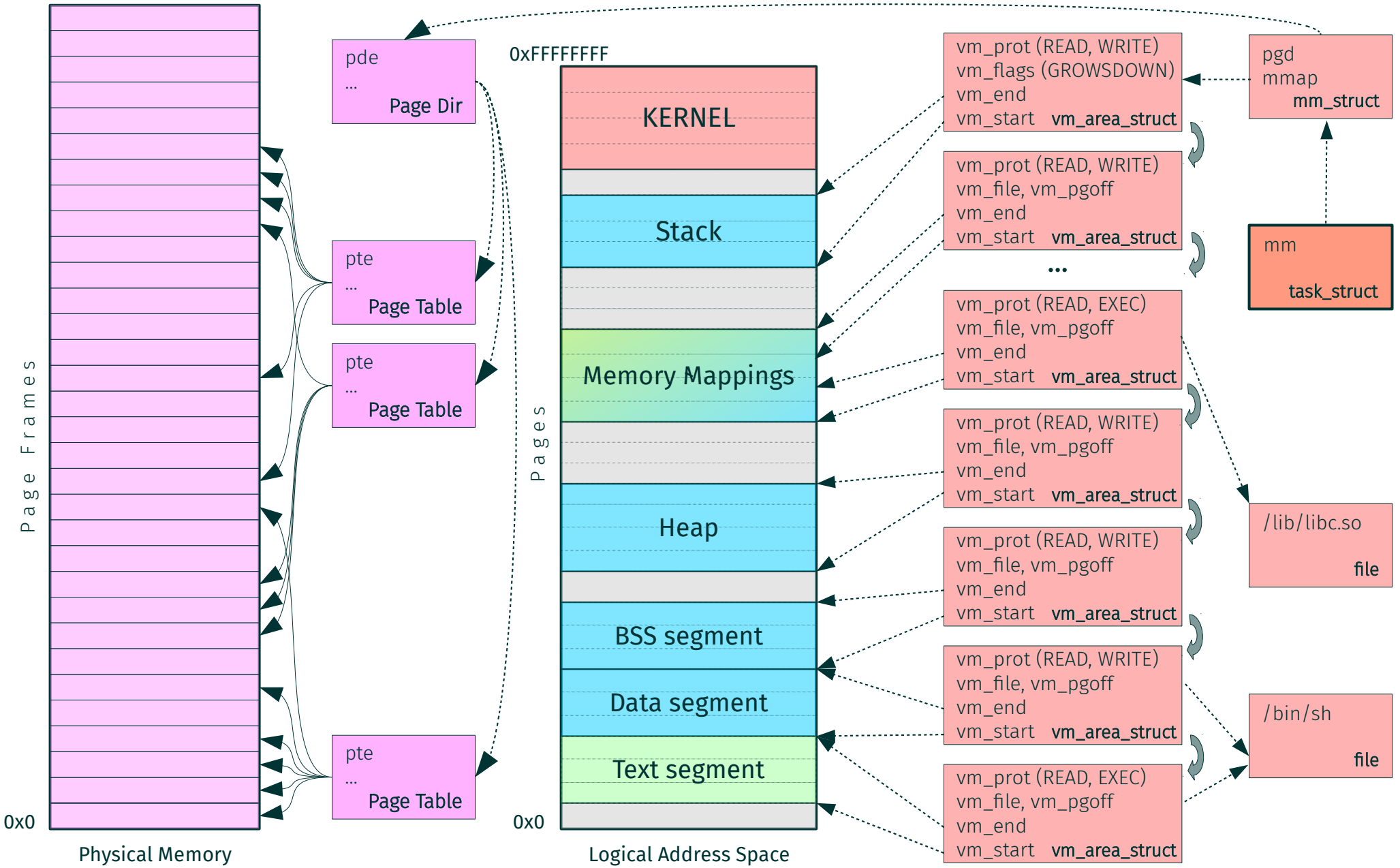
Mapovanie oblastí

- Systémové volanie **`mmap()`** vytvára nové mapovanie vo virtuálnom adresovom priestore procesu.
- **`void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);`**
  - Ak je **`addr`** rovné NULL, jadro zvolí vhodnú adresu.
  - **`prot`**: **`PROT_EXEC`** | **`PROT_READ`** | **`PROT_WRITE`** alebo **`PROT_NONE`**
  - **`flags`**: **`MAP_SHARED`** | **`MAP_PRIVATE`**, **`MAP_ANONYMOUS`**, **`MAP_HUGETLB`**, **`MAP_LOCKED`**, **`MAP_STACK`**, ...
  - Údaje použije OS pri nastavení PTE stránok tvoriacich túto oblasť.
- Vytvorí mapovanie od virtuálnej adresy ktorú vráti s dĺžkou **`length`**.
- Ak je zadaný deskriptor otvoreného súboru **`fd`**, pamäťová oblasť bude mapovaná na obsah súboru s posunutím **`offset`** od začiatku súboru.
- Ak ide o anonymné mapovanie, **`fd`** a **`offset`** sa ignorujú.
- Volanie **`munmap()`** zruší mapovanie, **`mremap()`** zmení (zmenší/zväčší/presunie) mapovanie.

Implementácia virtuálnej pamäti procesu



Virtuálna a fyzická pamäť

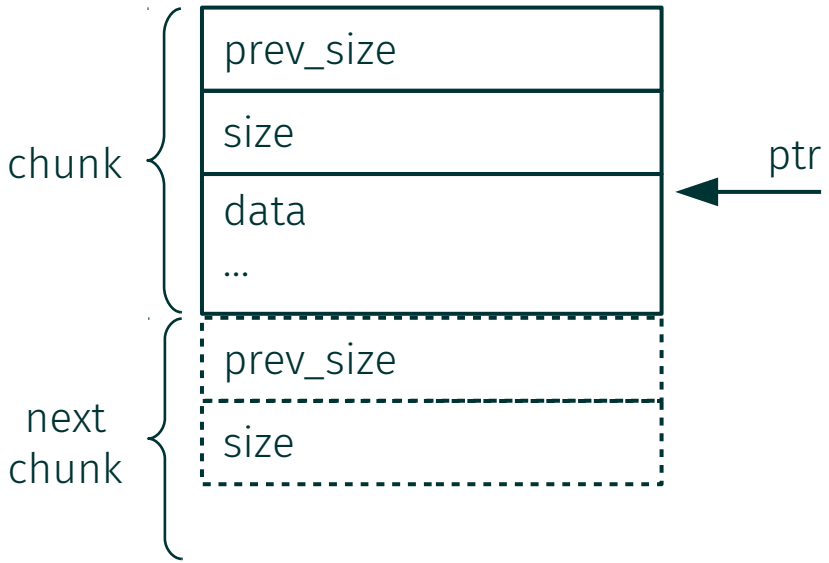


Overcommit

Knižničné funkcie malloc() a free()

- Knižničné funkcie *glibc* (**`stdlib.h`**).
- Funkcia **`malloc(size)`** alokuje size bajtov a vráti pointer na alokovanú pamäť, alebo NULL v prípade chyby. Funcia **`free(ptr)`** uvoľní pamäť alokovanú volaním **`malloc()`**.
- Knižnica uchováva zreťazený zoznam voľných blokov.
- Každý pointer vrátený funkciou **`malloc()`** ukazuje “do prostriedku” štruktúry (tzv. *chunk*).
- Napríklad:

```
char *ptr = malloc(256);
int size = *((int*)ptr - 1);
// size == 256
```



Funkcia malloc() a systémové volania

- Funkcia **`malloc()`** alokuje pamäť v časti *heap*, ktorej celkovú veľkosť – teda veľkosť dátového segmentu – je možné nastaviť systémovým volaním **`brk()`**.
- Pokiaľ je dosť voľnej pamäte v *heap-e*, funkcia **`malloc()`** sa vykoná **bez systémového volania**.
- Ak nie, posunie sa hranica dátového segmentu systémovým volaním: `int brk(void *end_data_segment);`
- Funkcia **`free()`** zníži veľkosť volaním **`sbrk()`** a tak vráti pamäť, ak na vrchole časti *heap* bude súvislý voľný blok veľkosti M_TRIM_TRESHOLD.
- Ak je veľkosť požadovanej pamäte väčšia než M_MMAP_TRESHOLD (128kB), funkcia **`malloc()`** použije na jej alokáciu systémové volanie **`mmap()`**.
  - Namapuje novú anonymnú oblasť.
- Parametre alokácie je možné nastaviť funciou **`mallopt()`**, napr. M_CHECK_ACTION. Kontrola konzistentnosti **`mcheck()`**.

Alokácia a dostupná pamäť

- Alokácia pamäte štandardne využíva optimistickú stratégiu.
  - Ukazuje sa, že nie všetky alokované oblasti procesy skutočne aj využívajú.
- Nealokovanie fyzickej pamäti pri mapovaní, ale až pri prvom prístupe, šetrí pamäť ktorá by inak bola nevyužitá.
- Ak funkcia `malloc()` vráti nenulový pointer neznamená to, že pamäť bude skutočne k dispozícii.
- Ak sa pri prístupe do pamäti zistí, že nie je možné uvoľniť žiadnu stránku, OS Linux (*OOM killer*) vyberie proces ktorý bude zrušený.
  - Toto správanie je v podstate nekorektné.
- Vlastnosti je možné nastavovať cez `/proc/sys/vm/overcommit_memory`.
  - **0**: heuristika, **1**: pamäť sa vždy pridelí, **2**: súčet veľkostí všetkých namapovaných oblastí nemôže prekročiť veľkosť odkladacieho priestoru (swapu) a polovicu fyzickej pamäte (alokovaná pamäť bude dostupná vždy).

Alokácia pamäti v jadre

SLOB/SLAB/SLUB alokátor

- Jadro poskytuje procesom pamäť vždy v celých stránkach.
  - Mapovanie oblasti je vždy zarovnané volaním **`mmap()`**.
  - Proces už spravuje pamäť sám, napr. *heap* spravuje **`malloc()`**.
- Jadro však pre svoju činnosť bežne potrebuje omnoho menšie časti pamäte, alokuje a uvoľňuje rôzne štruktúry typickej veľkosti.
- Aby sa znížila réžia spojená s alokáciou, jadro udržuje "predpripravené" objekty rôznej veľkosti (8, 16, 32, ..., 8k) a vopred inicializované štruktúry.
  - Rovnaké objekty sú umiestnené v po sebe idúcich stránkach, čo eliminuje fragmentáciu. Pole objektov rovnakej dĺžky sa nazýva *slab*, skupina *slab-ov* tvorí (*slab*) *cache*.
  - Každý modul (*driver*) môže mať vlastnú *cache*.
- Objekty zostávajú na mieste, ich pamäť netreba zakaždým alokovať, ani uvoľňovať. Zostávajú v inicializovanom stave. Rozhranie poskytujú funkcie **`kmalloc()`**/**`kfree()`**.
- Pre optimalizáciu prístupu sú objekty tiež zarovnané na veľkosť blokov HW (L1, L2) *cache*. Ich rovnomerné využitie sa dosahuje tzv. farbením, rôznym posunutím od začiatku stránky.

slabinfo

```
$ cat /proc/slabinfo
slabinfo - version: 2.1
# name            <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab> : tunables <limit>
<batchcount> <sharedfactor> : slabdata <active_slabs> <num_slabs> <sharedavail>
vm_area_struct     81290  89700    200   20    1 : tunables    0    0    0 : slabdata   4485   4485      0
mm_struct            780    780   1088   30    8 : tunables    0    0    0 : slabdata     26     26      0
files_cache          874    874    704   23    4 : tunables    0    0    0 : slabdata     38     38      0
task_struct         1059   1170   6144    5    8 : tunables    0    0    0 : slabdata    234    234      0
kmalloc-8k           173    184   8192    4    8 : tunables    0    0    0 : slabdata     46     46      0
kmalloc-4k          1029   1048   4096    8    8 : tunables    0    0    0 : slabdata    131    131      0
kmalloc-2k          1596   1632   2048   16    8 : tunables    0    0    0 : slabdata    102    102      0
kmalloc-1k          3088   3488   1024   32    8 : tunables    0    0    0 : slabdata    109    109      0
kmalloc-512         3484   4672    512   32    4 : tunables    0    0    0 : slabdata    146    146      0
kmalloc-256         4136   4320    256   32    2 : tunables    0    0    0 : slabdata    135    135      0
kmalloc-192         7424   7581    192   21    1 : tunables    0    0    0 : slabdata    361    361      0
kmalloc-128         4462   4672    128   32    1 : tunables    0    0    0 : slabdata    146    146      0
kmalloc-96          5568   6006     96   42    1 : tunables    0    0    0 : slabdata    143    143      0
kmalloc-64         27264  28864     64   64    1 : tunables    0    0    0 : slabdata    451    451      0
kmalloc-32         78408  80512     32  128    1 : tunables    0    0    0 : slabdata    629    629      0
kmalloc-16         71249  75520     16  256    1 : tunables    0    0    0 : slabdata    295    295      0
kmalloc-8         130478 132608      8  512    1 : tunables    0    0    0 : slabdata    259    259      0
```

Príklad: stránkovanie, výber obete

Príklad: Umiestnenie programu v pamäti

- Príklad: Veľkosť stránky je 1kB a proces má pridelené štyri stránkové rámy. Pred začatím vykonávania časti programu sa príslušné stránky nevyužívali. Koľko nastane výpadkov stránok pri vykonaní nasledujúcej časti programu pri použití algoritmu výberu obete LRU?
- `for (i=0; i<n; i++) { A[i] = B[i] + C[i]; }`

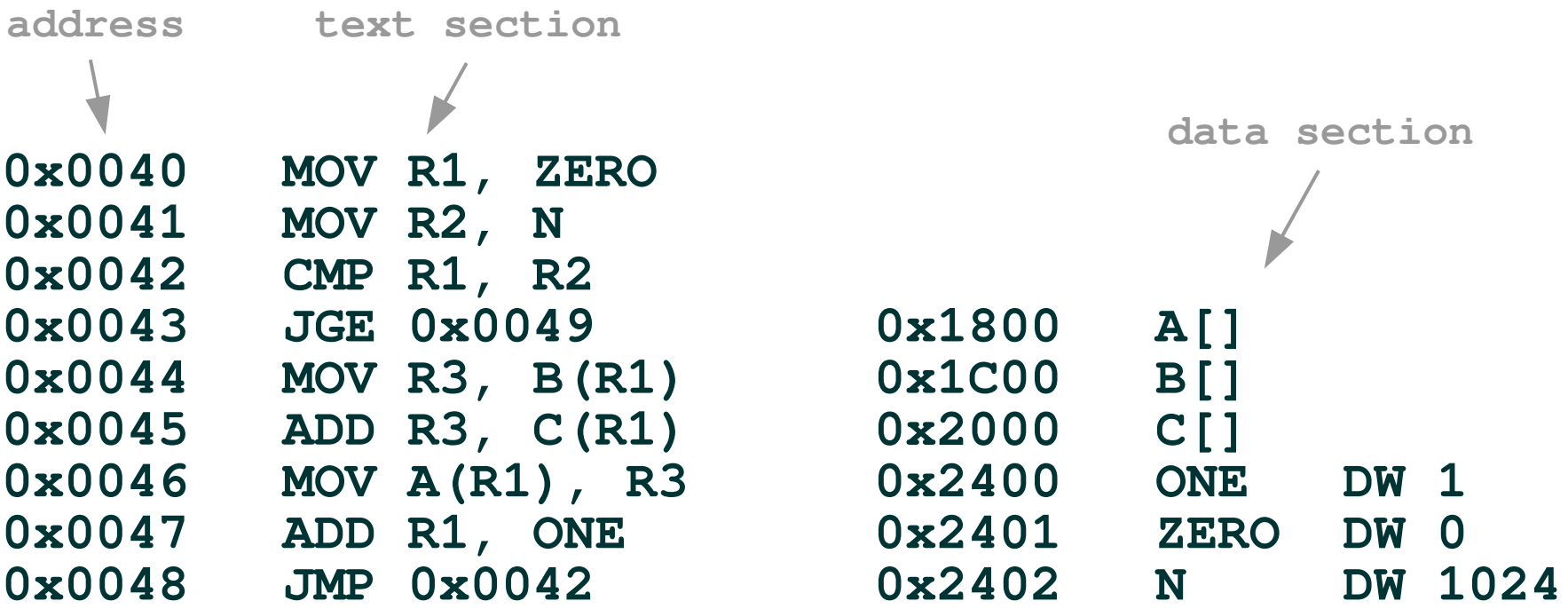
```
0x0040   MOV R1, ZERO
0x0041   MOV R2, N
0x0042   CMP R1, R2
0x0043   JGE 0x0049          0x1800   A[]
0x0044   MOV R3, B(R1)       0x1C00   B[]
0x0045   ADD R3, C(R1)       0x2000   C[]
0x0046   MOV A(R1), R3       0x2400   ONE    DW 1
0x0047   ADD R1, ONE         0x2401   ZERO   DW 0
0x0048   JMP 0x0042          0x2402   N      DW 1024
```

Príklad: Umiestnenie programu v pamäti

- Zápis programu vo vyššom jazyku (C):

```
for (i=0; i<n; i++) { A[i] = B[i] + C[i]; }
```

- Po preklade kompilátorom:

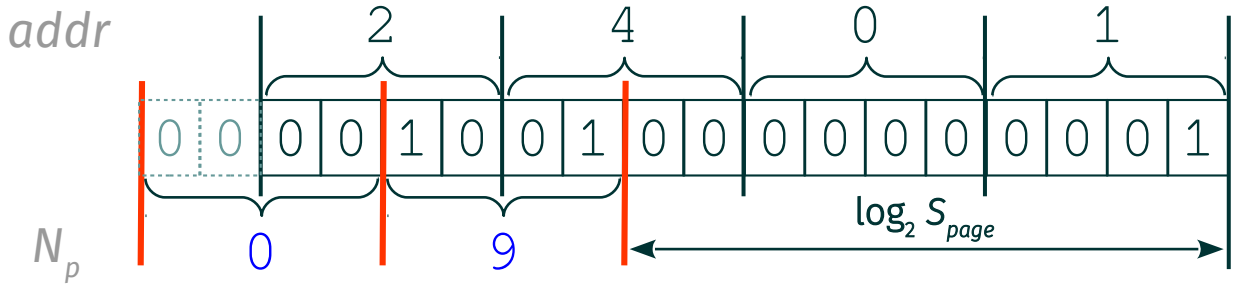


Príklad: Umiestnenie programu v pamäti

- Príklad: Veľkosť stránky je 1kB a proces má pridelené štyri stránkové rámy. Pred začatím vykonávania časti programu sa príslušné stránky nevyužívali. Koľko nastane výpadkov stránok pri vykonaní nasledujúcej časti programu pri použití algoritmu výberu obete LRU?
- Postup:
  - Zistíme to "simuláciou" vykonávania programu a algoritmu výberu obete.
  - Aby sme zistili výpadky, musíme vedieť ktoré stránky sú v pamäti a ku ktorým stránkam sa pristupuje.
  - Potrebujeme poznať postupnosť odkazov na stránky – *reference string*.
  - Na to potrebujeme poznať v ktorých stránkach ležia adresy ktoré program využíva.

Príklad: Výpočet čísla stránky

- Veľkosť stránky: $S_{page} = 1kB$, $n_{page} = \log_2 S_{page}$
- Číslo stránky: $N_p(addr) = addr \,/\, S_{page} = \text{shr } addr, n_{page}$




```
0x0040   MOV R1, ZERO
0x0041   MOV R2, N
0x0042   CMP R1, R2
0x0043   JGE 0x0049
0x0044   MOV R3, B(R1)
0x0045   ADD R3, C(R1)
0x0046   MOV A(R1), R3
0x0047   ADD R1, ONE
0x0048   JMP 0x0042
```

```
0x1800   A[]
0x1C00   B[]
0x2000   C[]
0x2400   ONE    DW 1
0x2401   ZERO   DW 0
0x2402   N      DW 1024
```

Príklad: Postupnosť odkazov na stránky

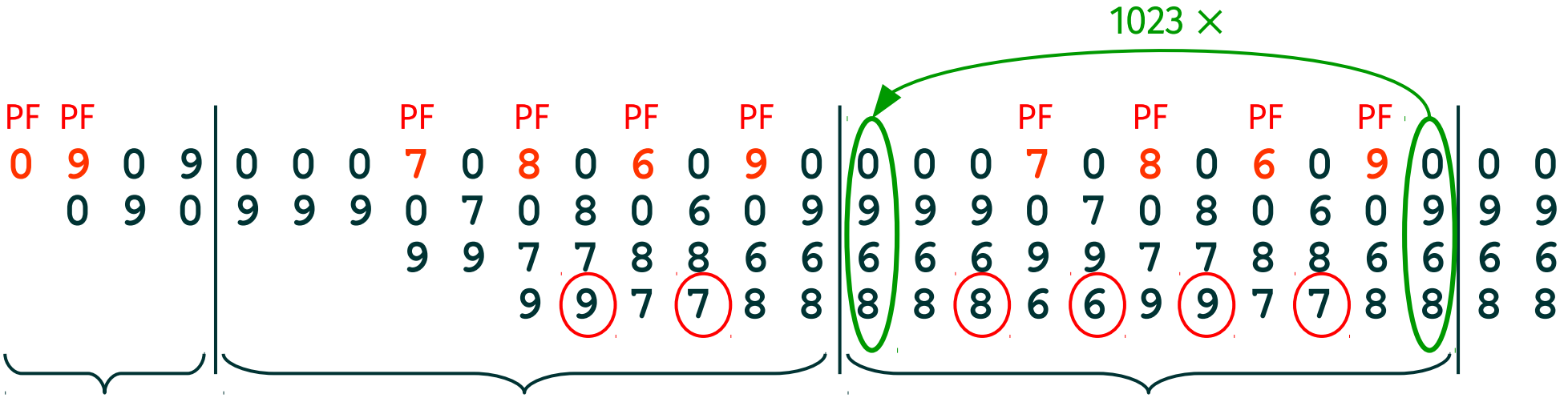
- Reference string:
  - Pri vykonávaní uvedenej časti programu bude procesor postupne robiť prístup k nasledujúcim stránkam.
- 0, 9, 0, 9, (0, 0, 0, 7, 0, 8, 0, 6, 0, 9, 0)^{1024}, 0, 0

```
0  0x0040   MOV R1, ZERO    0,9
   0x0041   MOV R2, N       0,9
   0x0042   CMP R1, R2      0
   0x0043   JGE 0x0049      0
   0x0044   MOV R3, B(R1)   0,7
   0x0045   ADD R3, C(R1)   0,8
   0x0046   MOV A(R1), R3   0,6
   0x0047   ADD R1, ONE     0,9
   0x0048   JMP 0x0042      0

6  0x1800   A[]
   ...
7  0x1C00   B[]
   ...
8  0x2000   C[]
   ...
9  0x2400   ONE
   0x2401   ZERO
   0x2402   N
```

Príklad: Simulácia algoritmu LRU

- Reference string:
- 0, 9, 0, 9, (0, 0, 0, 7, 0, 8, 0, 6, 0, 9, 0)^{1024}, 0, 0



2 PF + 4 PF + 1023 · 4 PF = 4098 PF

■

To je zatiaľ všetko