

Základy OS

Struktura OS

- správa procesů
- správa paměti
- správa souborů
- správa I/O
- bezpečnost

Základní definice

- **OS** = SW vrstva pro správu HW poskytující jednotné rozhraní programům k HW
- **program** = spustitelný kód (sekvence instrukcí)
- **proces** = instance běžícího programu
- **vlákno** = 1 proces = N vláken – vlákna vykonávají instrukce

Dělení OS

- **dle sdílení CPU:**
 - jednoprocesorový
 - multiprocesorový
- **dle typu interakce:**
 - dávkový systém
 - interaktivní

OS reálného času

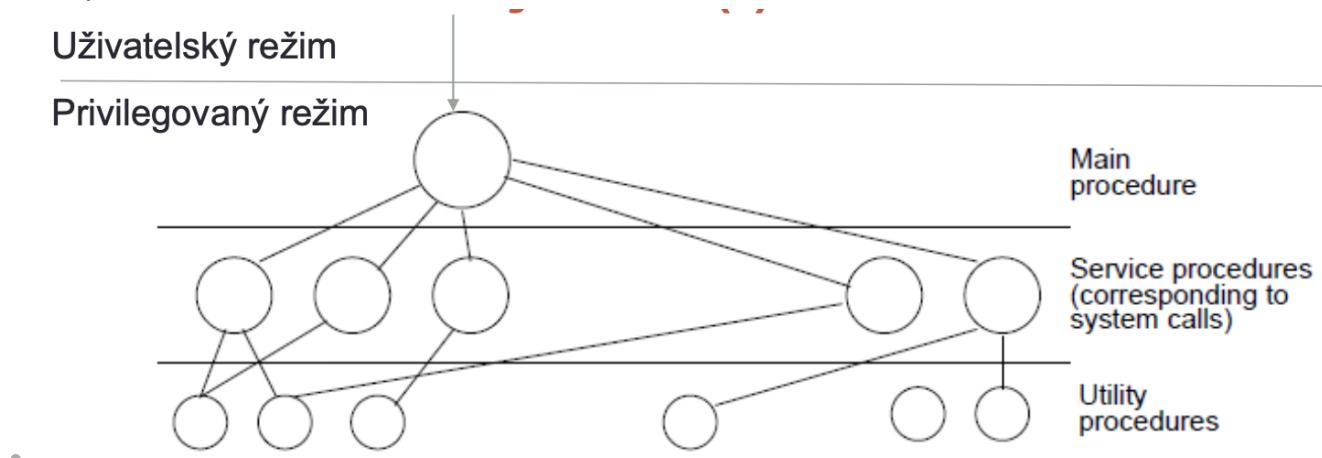
- výsledek musí být získán v omezeném čase
- přísné požadavky na čas odpovědí
- **typy:**
 - časově ohraničené požadavky
 - nejlepší snaha

Architektury OS

- **OS** = jádro + systémové zdroje
- **dělení:**
 - **monolitické jádro** = jádro je 1 funkční celek
 - **mikrojádro** = malé jádro, oddělitelné části pracují jako procesy
 - **hybridní jádro** = kombinace

Monolitické jádro

- 1 spustitelný soubor
- uvnitř moduly pro jednotlivé funkce
- jeden program
- řízení se provádí voláním podprogramů
- např. UNIX, Linux, MS DOS



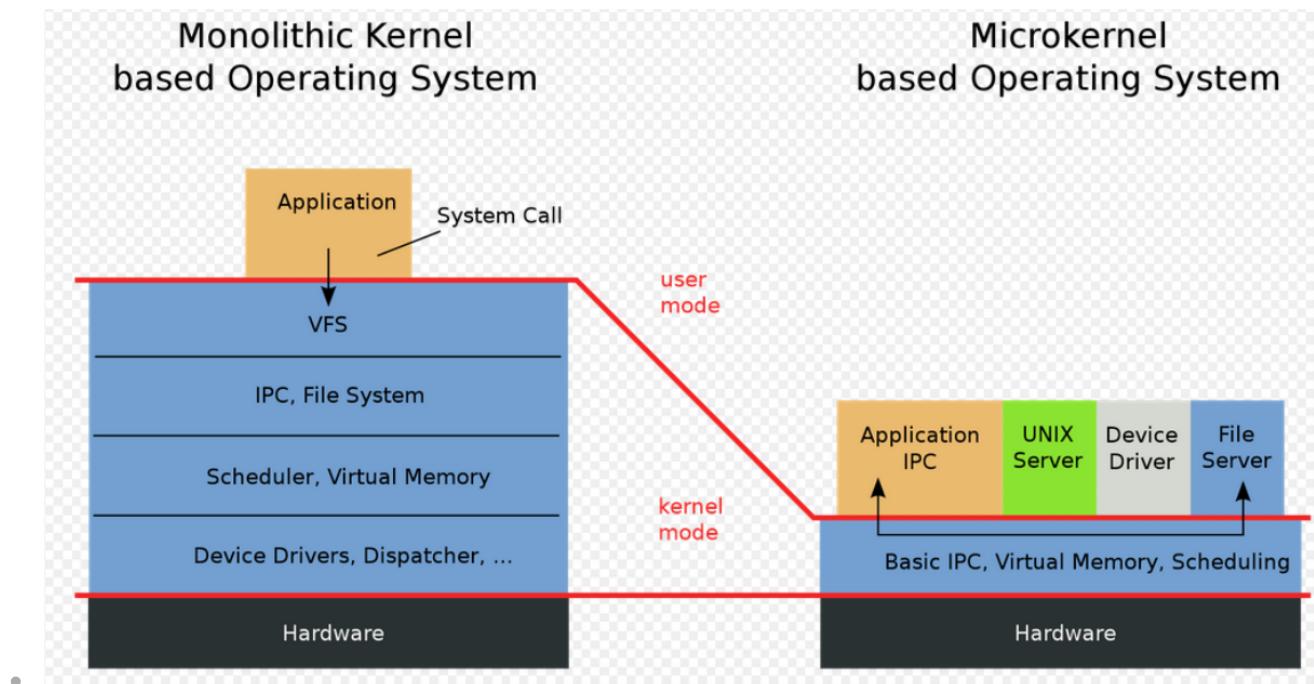
- **main procedure** = vstupní bod jádra
- **service procedures** = systémová volání
- **utility procedures** = pomocné procedury pro systémová volání

Mikrojádro

- model **klient-server**

- většinu činností OS vykonávají procesy mimo jádro (servery)
- **mikrojádro:**
 - poskytuje pouze nejdůležitější nízkoúrovňové funkce
 - běží v privilegovaném režimu

Monolitické jádro X Mikrojádro



Privilegovaný a uživatelský režim procesory

- režim určen podle mode bitu

Privilegovaný režim

- běží v něm jádro OS
- všechny instrukce povoleny

Uživatelský režim

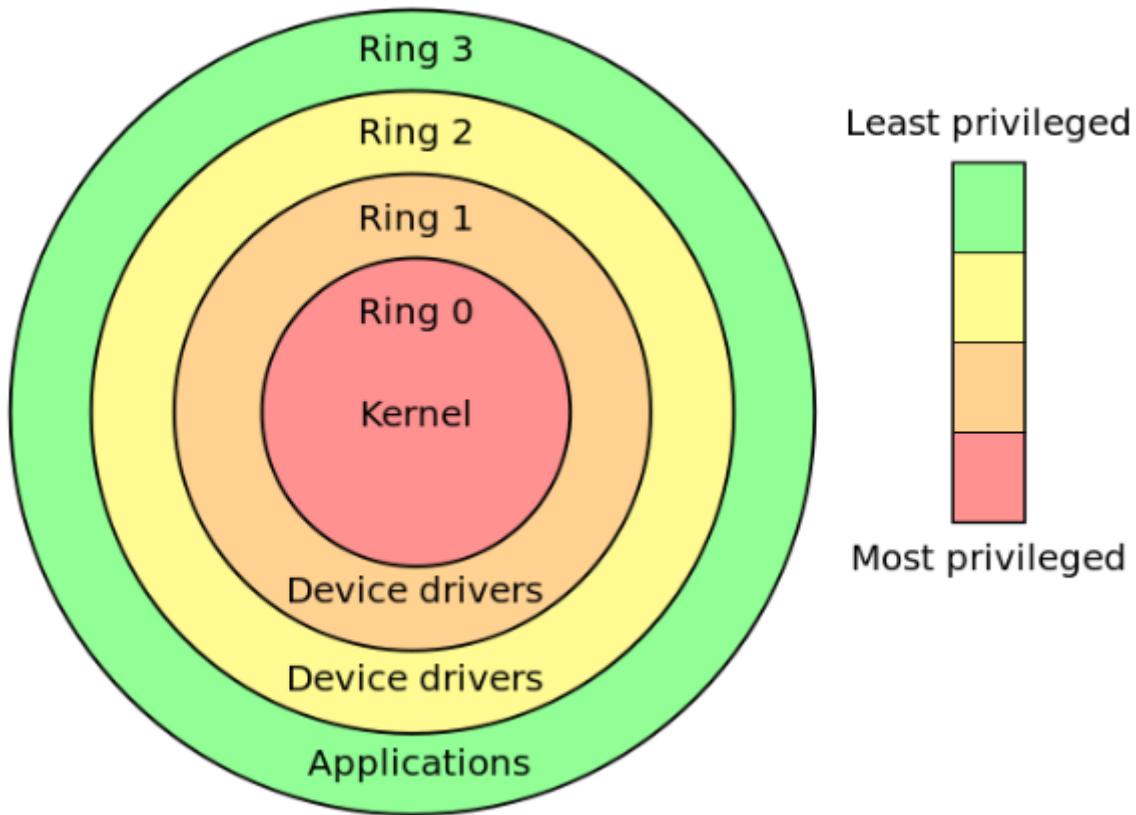
- běží v něm aplikace
- některé instrukce zakázány
- aplikace nemá přímý přístup k HW

Přepínání mezi režimy

- SW přerušení – INT
- speciální instrukce – sysenter/sysexit , syscall/sysret

Protection ring

- více stupňů ochrany než jen privilegovaný a uživatelský režim



- - Ring 0 = jádro
 - Ring 3 = aplikace

Systémové volání

- **systémové volání** = mechanismus pomocí, kterého aplikace volají služby OS
- **důvod:** aplikace nemají přímý přístup k HW (bezpečnostní prvek)
- **způsoby volání:**
 - napřímo
 - knihovní funkce

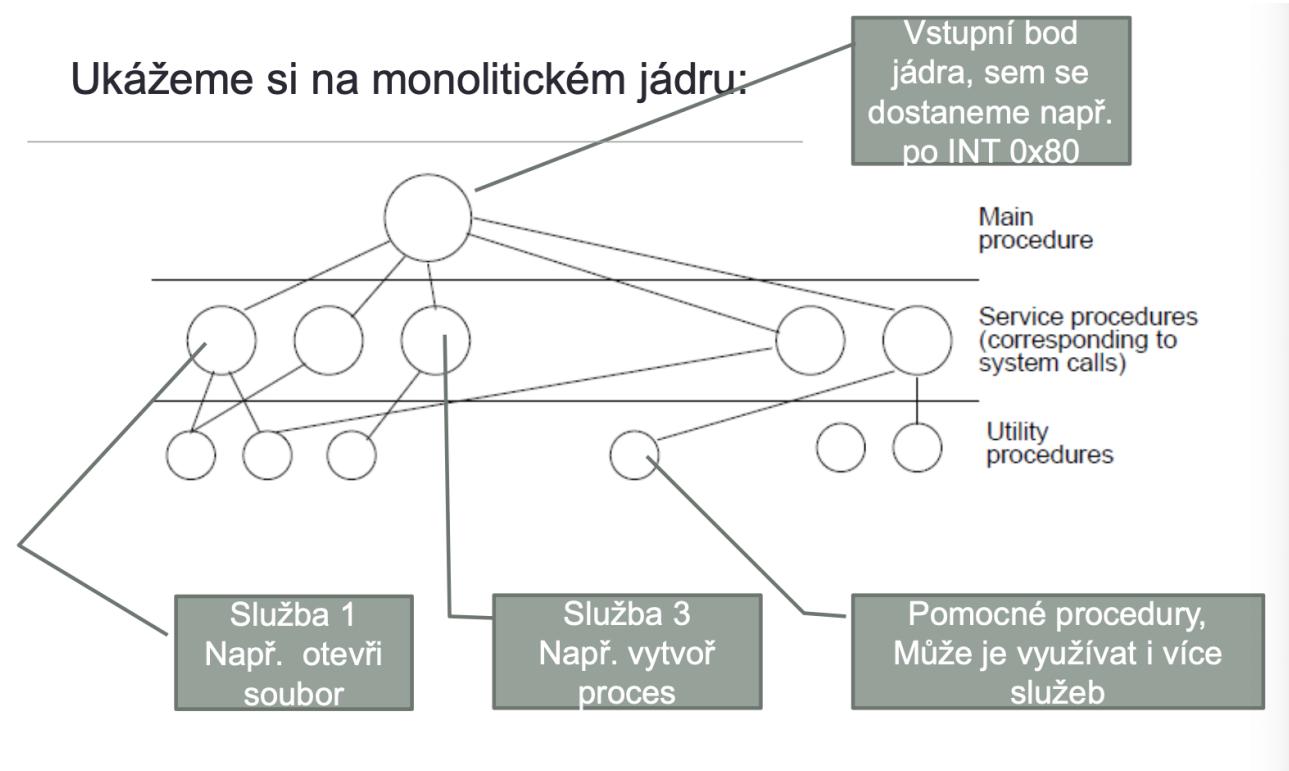
Příklad

1. do registru uloženo číslo volané služby
2. do dalších registrů/na zásobník uloženy parametry
3. přepnutí do režimu jádra
4. zpracování služby
5. návrat

Možnosti volání služeb jádra

1. instrukce INT 0x80
2. syscall(číslo_funkce)
3. použití wrapper funkce

Implementace služeb monolitického jádra



Přerušení

- **přerušení** = událost, kterou je potřeba obsloužit
- **obsluha přerušení** = obsluha události
- **princip:**

1. procesor přestane vykonávat sled instrukcí
2. vykoná obsluhu přerušení
3. pokračuje ve vykonávání sledu instrukcí

Druhy přerušení

HW přerušení (vnější)

- vyvolá I/O zařízení
- asynchronní
- maskovatelné

Vnitřní přerušení (výjimky)

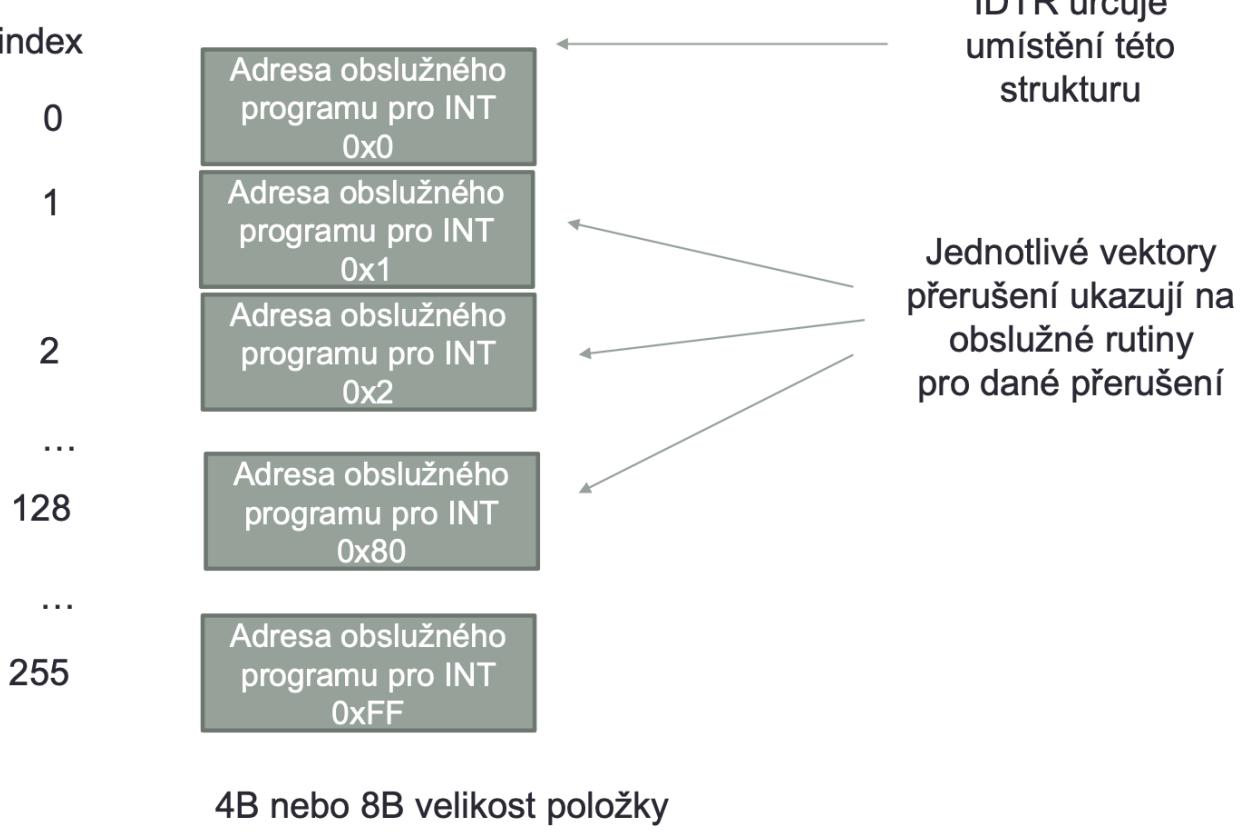
- vyvolává procesor
- synchronní
- nemaskovatelné
- např. dělení nulou, neplatná instrukce, výpadek stránky paměti
- **výpadek stránky paměti:**
 - v logickém adresním prostoru se odkazuje na stránku, která není namapovaná do RAM

SW přerušení

- vyvolá speciální strojová instrukce
- synchronní
- nemaskovatelné

Tabulka vektorů přerušení

- **tabulka vektorů přerušení** = pole, kde je na daném indexu adresa obsluhy přerušení
- **Reálný režim CPU:**
 - od adresy 0 až 1 KB v RAM
 - 256 x 4 bytový ukazatel
- **Protected mód CPU:**
 - od adresy **IDTR**
 - 256 x 8 bytový ukazatel



Obsluha přerušení

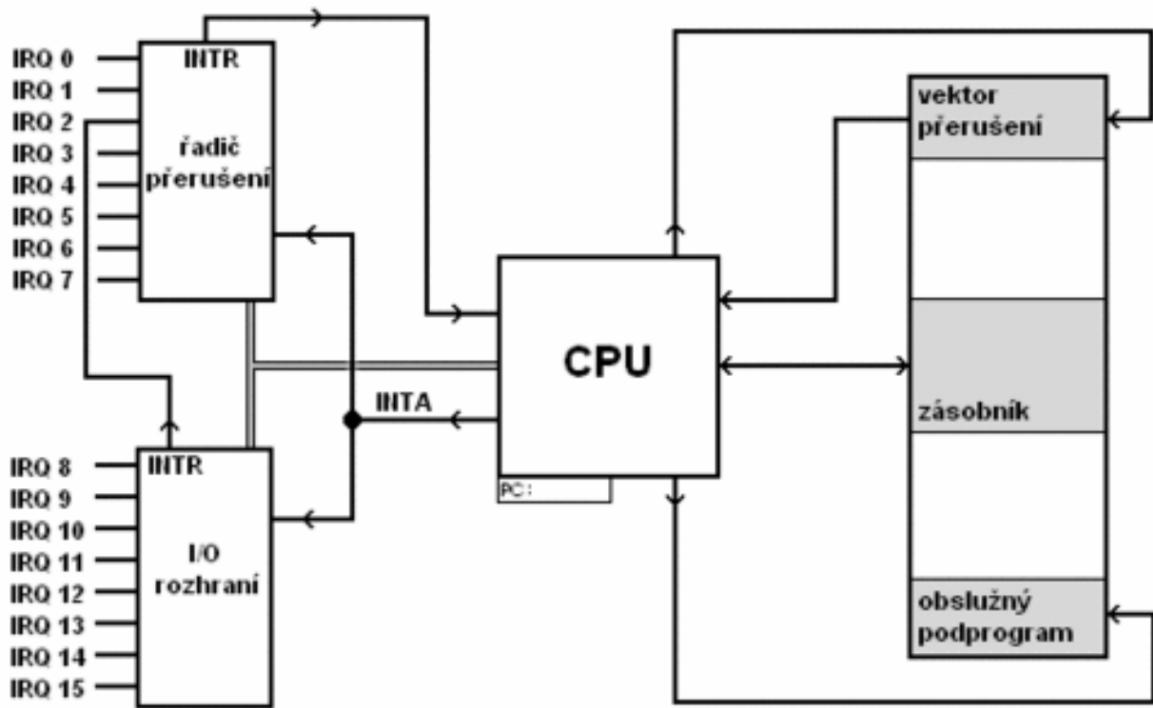
- mechanismus vyvolání přerušení (instrukce `INT x`)
 - na zásobník uložen registr FLAGS
 - zákaz přerušení (příznak IF)
 - na zásobník uložena hodnota kam bude vracet (PC – Program Counter)
- vykonání kódu obsluhy přerušení
- návrat z přerušení (instrukce `IRET`)
 - za zásobníku načtena návratová adresa (PC)
 - ze zásobníku obnoven registr FLAGS

Obsluha HW přerušení

- IRQ (Interrupt Request)** = signál, kterým zařízení (klávesnice) žádá CPU o přerušení aktuálního sledu instrukcí a obsluhu daného přerušení
- IRQL (Interrupt Request Level)** = priorita přerušovacího požadavku
- obsluha:**
 - zařízení sdělí řadiči přerušení, že nastalo přerušení
 - řadič upozorní CPU, že jsou pending přerušení
 - CPU dokončí rozpracovanou instrukci
 - CPU zjistí od řadiče jaké přerušení má nejvyšší IRQL

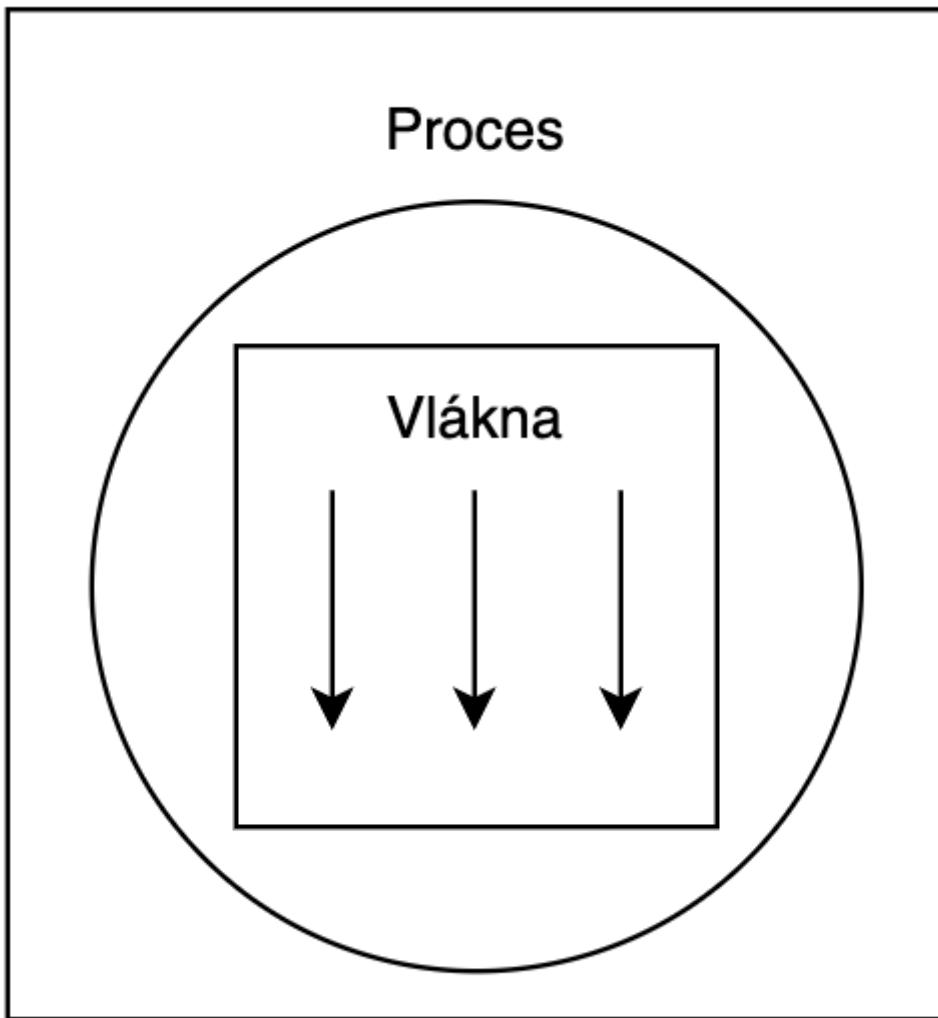
5. CPU dostane od řadiče přes datovou sběrnici index do tabulky přerušení
6. CPU spustí obsluhu
7. CPU uloží FLAGS a PC, vykoná obsluhu a informuje řadiče o dokončení a následně pokračuje ve sledu instrukcí

- **řadič přerušení:**



Správa procesů

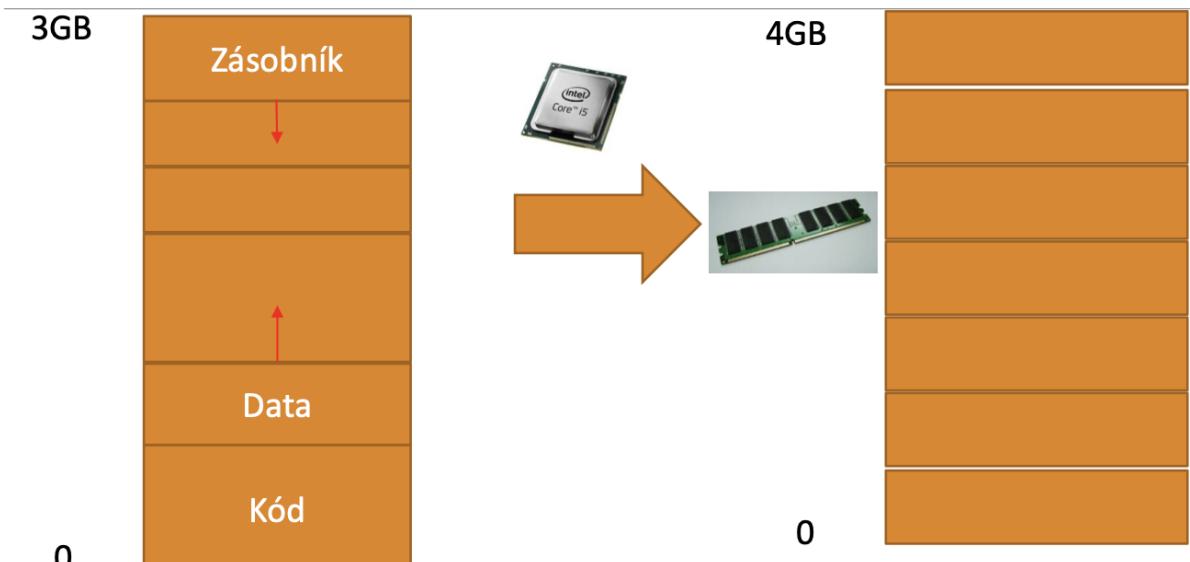
- **proces** = instance běžícího programu



•

Adresní prostor procesu

- virtuální paměť (od 0 do nějaké adresy) namapována do fyzické paměti (RAM)
- **MMU (Memory Management Unit)** – zajišťuje mapování
 - součást CPU
- kód programu, zásobník, data
- **stavové informace** = registry a další informace potřebné k běhu procesu



Virtuální adresy

- používá proces

Fyzická paměť RAM

Fyzické adresy

Do ní se mapují data procesů, jádra, ...

Registry

- **SP** = stack pointer
- **BP** = práce se zásobníkem
- **SI** = offset zdroje
- **DI** = offset cíle
- **CS** = code segment (kód)
- **DS** = data segment (data)
- **SS** = stack segment (zásobník)
- **speciální:**
 - **IP** = offset vykonávané instrukce vůči CS
 - **FLAGS**
 - **IF** = interrupt flag (povolení přerušení)
 - **ZF** = zero flag (je-li výsledek operace 0)

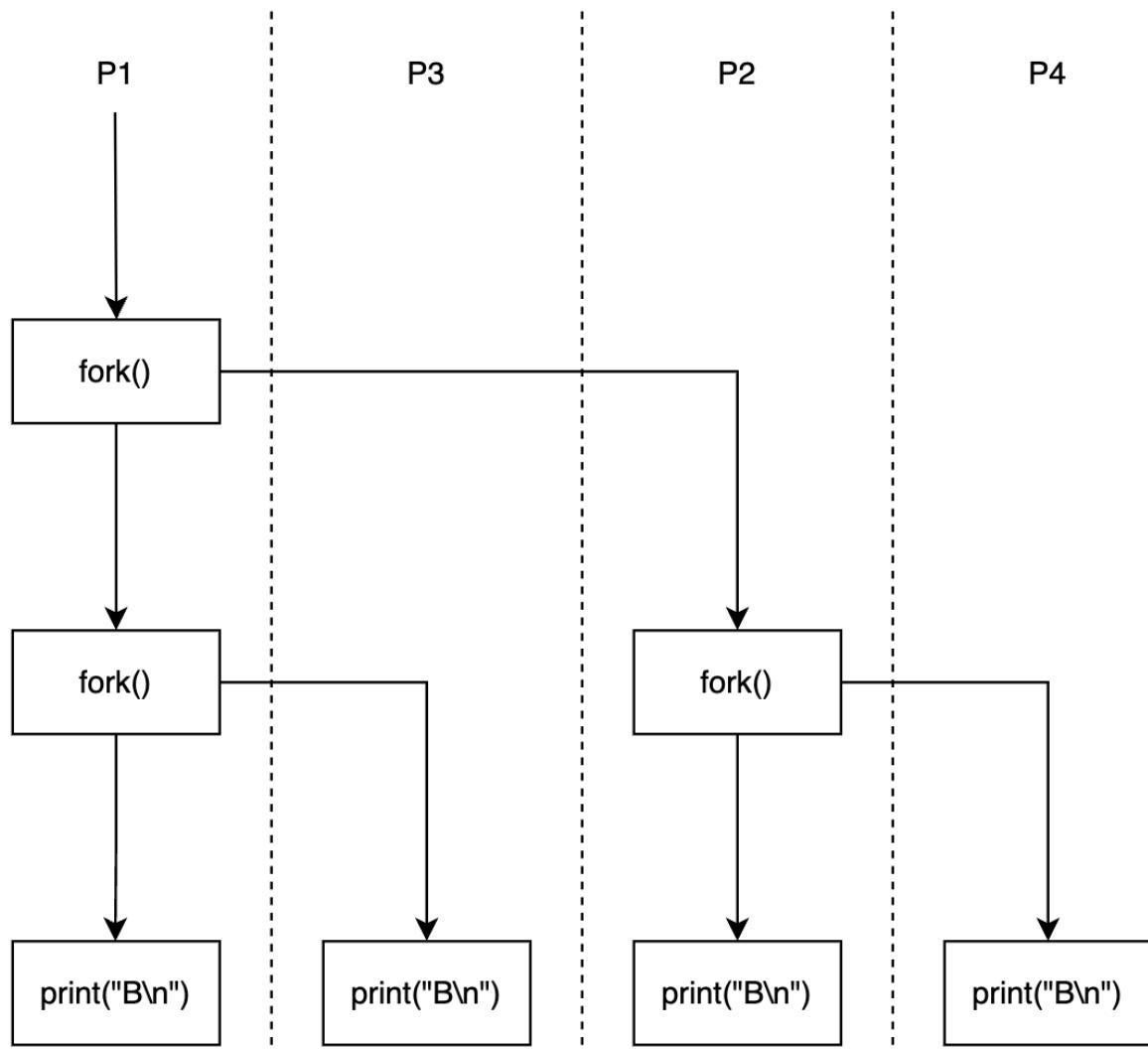
fork()

- vytvoří klon původního procesu
- nový proces vykonává stejný kód

- nový proces má jiné PID
- **návratová hodnota:**
 - `int pid = fork()`
 - `pid == 0 => potomek`
 - `pid > 0 => rodič`

Graf procesů

```
fork();
fork();
printf("B \n");
```

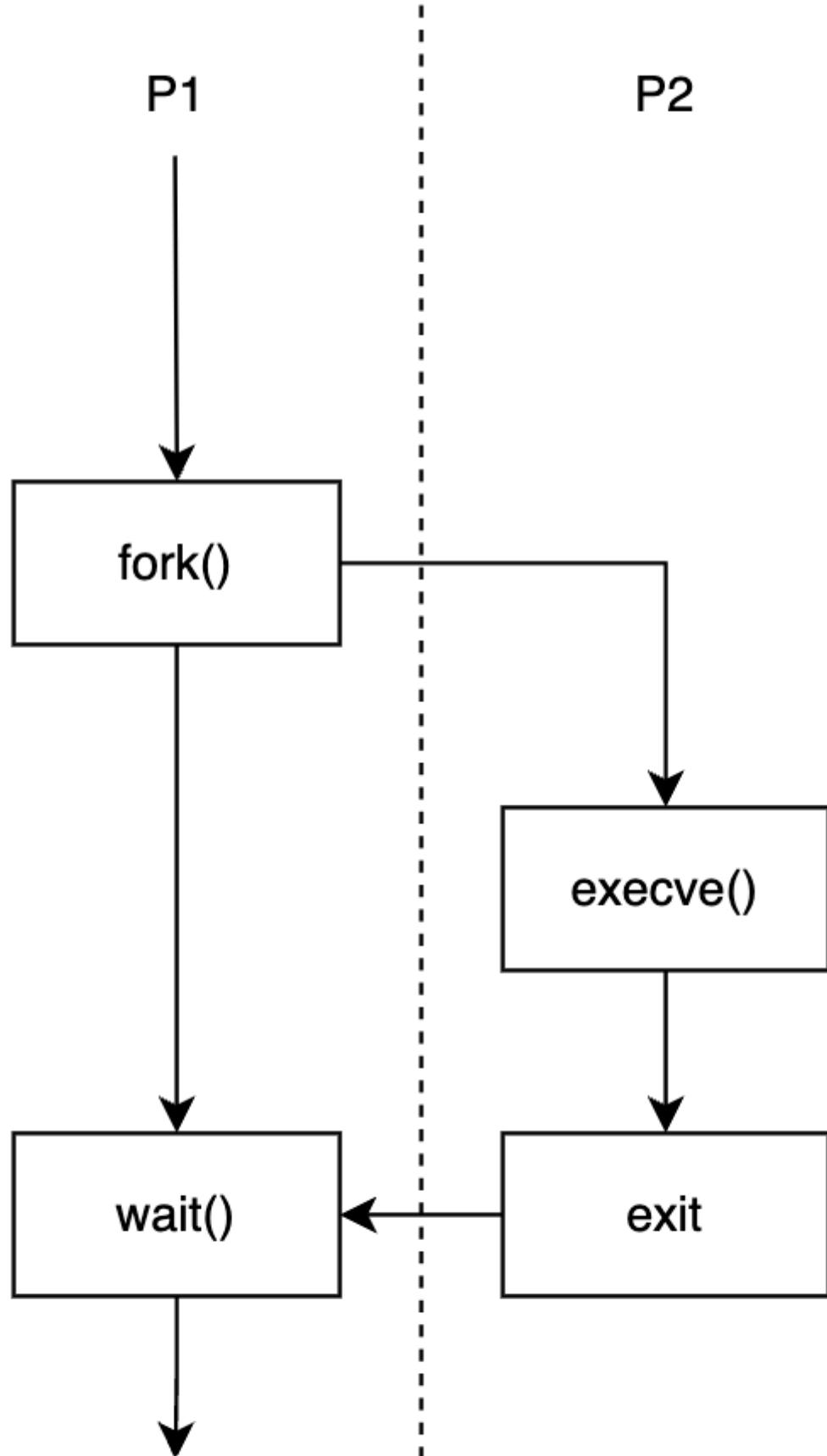


•

`execve()`

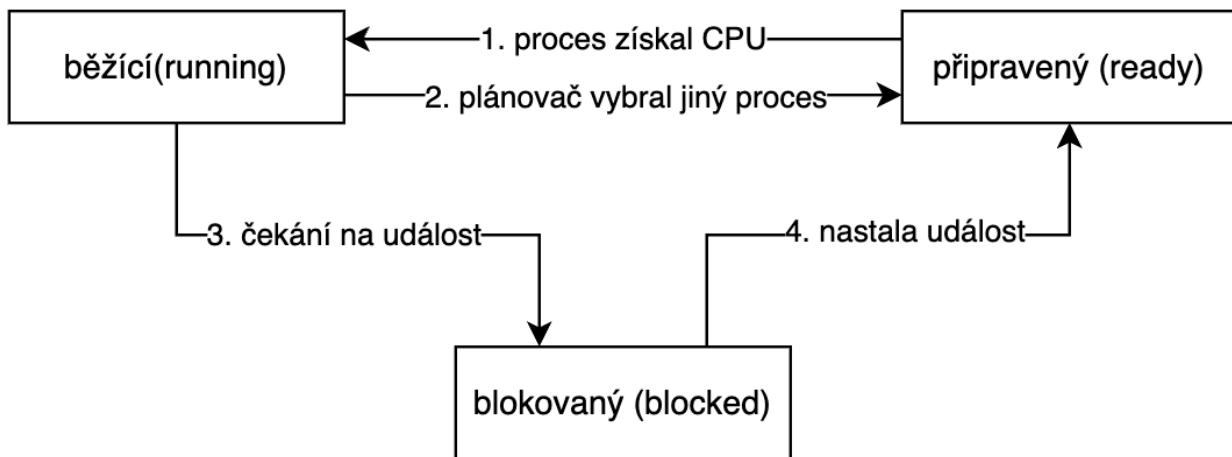
- proces začne vykonávat jiný program

```
if (fork() == 0)
    execve("bin/ls", argv, envp);
else
    wait(NULL);
```



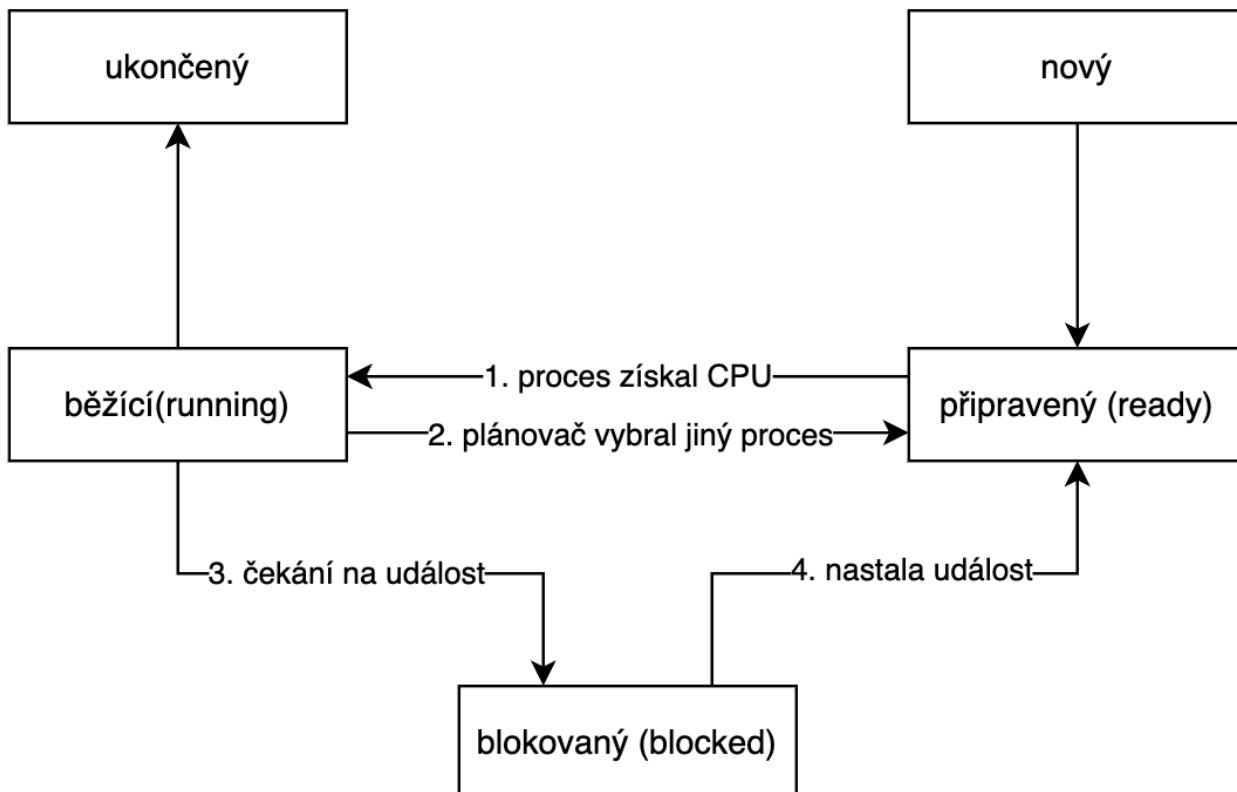
Základní stavy procesů

- **běžící** = vykonává instrukce
- **připravný** = dočasně pozastaven aby mohl jiný proces vykonávat instrukce
- **blokovaný** = neschopný běhu dokud nenastane externí událost



Dodatečné stavy procesů

- **nový** = nově vytvořený proces, který přejde ze stavu nový do připravený
- **ukočený** = proces, který skončil, přechází ze stavu běžící do ukončený



Dodatečné Linux stavy

- **zombie** = proces dokončil svůj kód, ale stále má záznam v tabulce procesů
- **sirotek** = kód procesu stále běží, ale skončil rodičovský proces

Ukončení procesu

1. proces úspěšně vykonal kód
2. proces překročí limit nějakého zdroje
3. proces ukončí uživatel

Přepnutí procesu

- systém použije HW přerušení od časovače
- **pravidelná přerušení:**
 - zkонтroluje zda má proces čas běžet
 - ANO: návrat z přerušení a neděje se nic
 - NE: přeplánování na jiný proces = **přepnutí kontextu**

Přepnutí kontextu

1. uloží obsah registrů na stack
2. plánovač nastaví proces, který opouští CPU jako **ready**
3. plánovač vybere nový proces
4. plánovač nastaví mapu paměti nového procesu
5. plánovač nastaví zásobník, načtě z něj obsah registrů
6. provede se návrat z přerušení **IRET**

Tabulka procesů

- evidence existujících procesů
- záznamy = PCB (Proces Control Block)
- umožňuje plánovači rozhodnout, který proces bude běžet

PCB (Process Control Block)

- **PCB** = záznam v tabulce procesů
- **obsahuje:** informace potřebné pro opětovné spuštění proces

MMU – Memory Management Unit

- řeší problém více procesů v paměti
 - každý proces si myslí, že má paměť pro sebe
 - proces nesmí zasahovat do paměti jiného
- **princip:**
 - mezi CPU a pamětí je MMU
 - program pracuje s virtuálními adresami
 - MMU je převádí na fyzické
 - MMU je uvnitř CPU

Precedenční grafy

- **precedenční grafy** = popis pro vyjádření relací mezi procesy
- acyklický orientovaný graf
- **hrana grafu** = běh procesu p_i
- **spojení hran** = vztah mezi procesy
 - sériové = $S(a,b)$
 - paralelní = $P(a,b)$

Abstraktní primitiva

- **cobegin, coend**

```
cobegin
    C_1 || C_2 || ... || C_n
coend
```

Plánování procesů

- **podle času:**

- **krátkodobé** = CPU scheduling
 - kterému z připravených procesů bude přidělen procesor
- **střednědobé** = swap out
 - odsun procesu z vnitřní paměti na disk
- **dlouhodobé** = job scheduling
 - výběr, která úloha bude spuštěna

- **podle preemptivnosti:**

- **nepreemptivní** = pouze přechod běžící => blokovaný
- **preemptivní** = navíc i přechod běžící => připravený

- **dávkové systémy:**

- **průchodnost** = počet úloh dokončených za jednotku času
- **průměrná doba obrátky** = doba od zadání úlohy do systému a do dokončení úlohy
- hledáme optimální hodnotu průchodnosti a průměrné doby obrátky

- **interaktivní systémy:**

- **chceme:** minimalizaci doby na odpověď
- **je třeba dbát na:** efektivitu

- **RT systémy:**

- dodržování deadlines

Plánovač X Dispatcher

- **plánovač** = určí, který proces (vlákno) by nyní měl běžet
- **dispatcher** = provede přepnutí z aktuálního běžícího procesu na nově vybraný proces

Nepreemptivní plánování

- nemá přechod běžící => připravený
- každý proces **dokončí svůj CPU burst**
- proces má kontrolu nad CPU dokud se jí nevzdá
- vhodné pro dávkové systémy

Preemptivní plánování

- má přechod běžící => připravený
- kritické sekce i na jednojádru
- proces lze **kdykoliv přerušit během CPU burstu**
- větší režie
- vyžaduje **timer (časovač)**

Cíle plánování – obecné

- **spravedlivost** = srovnatelné procesy srovnatelně obsloužené
- **vynucení politiky** = dodržování stanovených pravidel
- **balance** = snaha vytížit všechny části systému
- **nízká režie plánování** = věnovat vyýkon procesům

Cíle plánování – dávkové systémy

- **propustnost** = maximalizovat počet jobů za jednotku času
- **doba obrátky** = minimalizovat čas mezi přijetím úlohy a jejím dokončením
- **CPU využití** = snaha mít CPU pořád vytížené

Plánovač

- **rozhodovací mód** = okamžik, kdy jsou vyhodnoceny priority procesů a vybrán proces pro běh
 - nepreemptivní
 - preemptivní
- **prioritní funkce** = určí prioritu procesu
 - většinou dvě složka – **statická + dynamická**
 - **statická** – přiřazena při startu procesu
 - **dynamická** = mění se dle chování procesu
 - **v kvantově orientovaných algoritmech:**
 - **priorita** = $1/f$
 - f = velikost časového kvanta, kterou proces naposledy použil
- **rozhodovací pravidla** = jak rozhodnout při stejně prioritě
 - **malá ppst stejné priority** => náhodný výběr
 - **velká ppst stejné priority** => cyklické přidělování kvanta, FIFO

Plánování procesů v dávkových systémech

- **FCFS** = First Come First Served

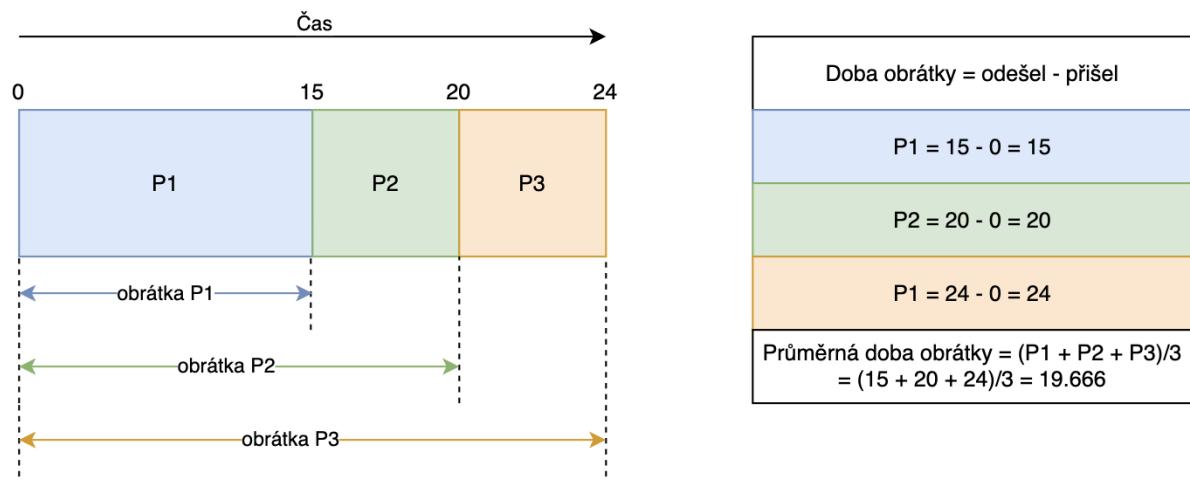
- **SJF** = Shortest Job First
- **SRT** = Shortest Remaining Time
- **Multilevel Feedback**

FCFS (First Come First Server)

- **nepreemptivní FIFO**
- **příklad:**

V čase nula budou v systému procesy P1, P2, P3 přišlé v tomto pořadí:

proces	Doba trvání (s)
P1	15
P2	5
P3	4

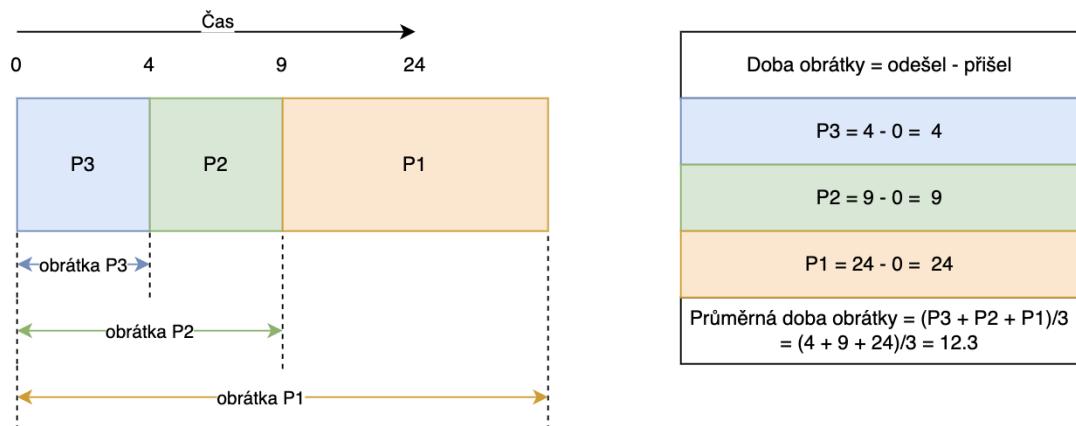


SJF (Shortest Job First)

- **nejkratší úloha jako první**
- **nepreemptivní**
- optimalizuje průměrnou dobu obrátky
- **příklad:**

V čase nula budou v systému procesy P1, P2, P3 příšlé v tomto pořadí:

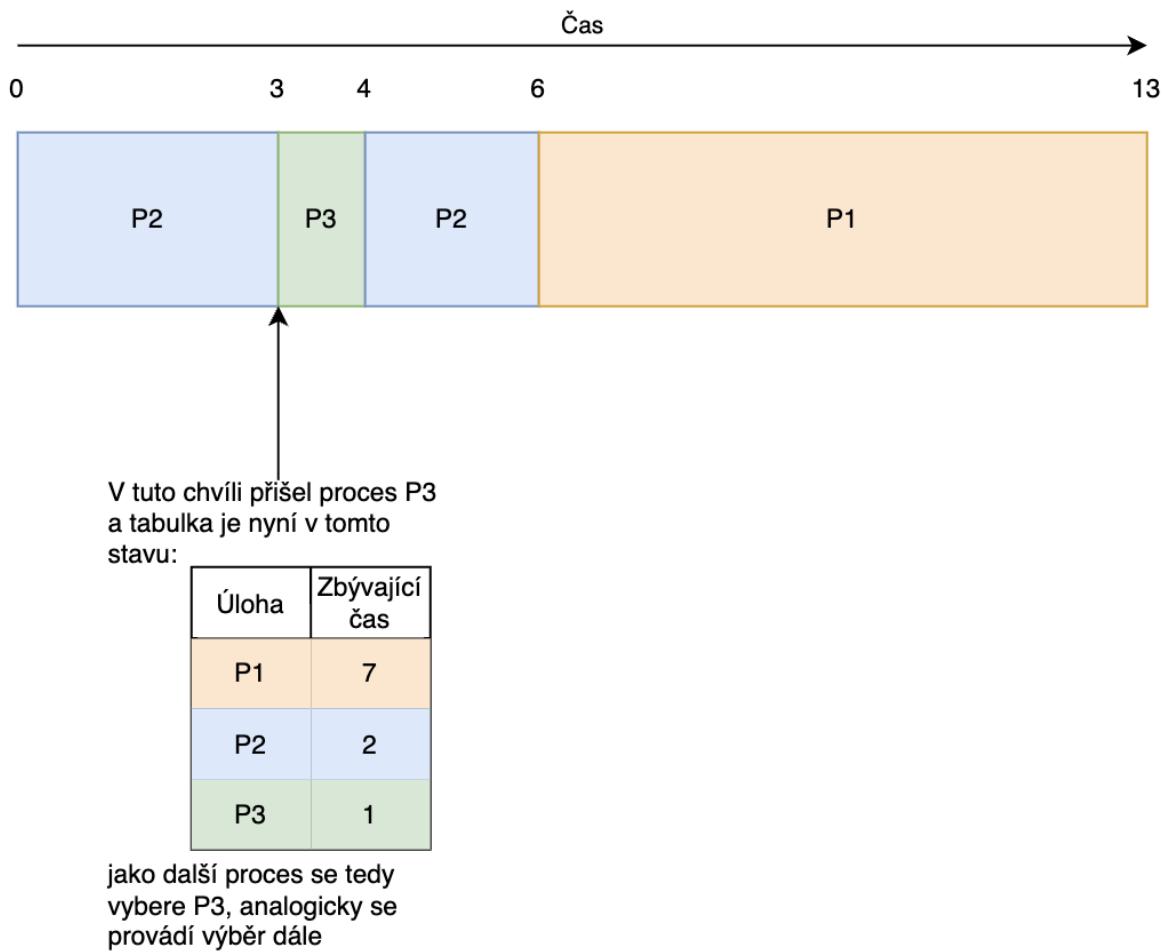
proces	Doba trvání (s)
P1	15
P2	5
P3	4



SRT (Shortest Remaining Time)

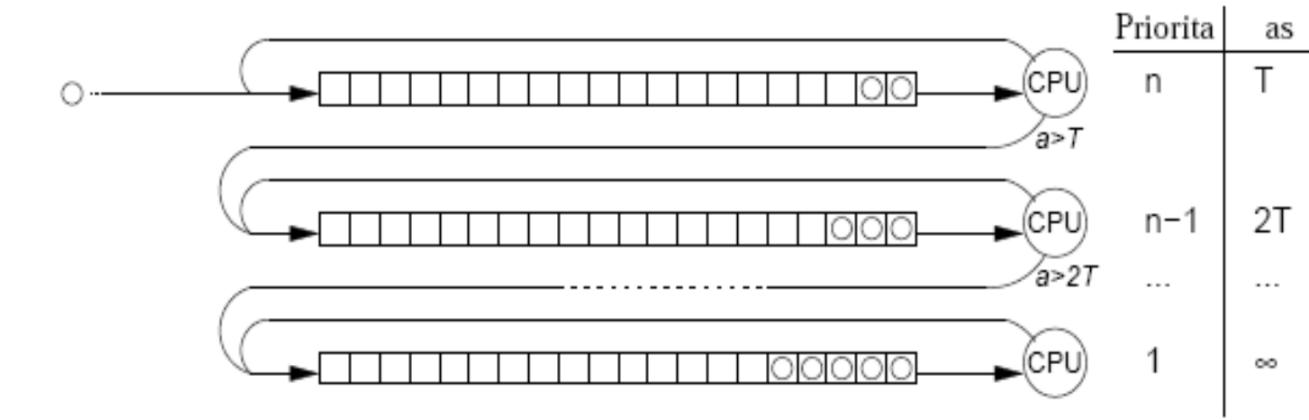
- plánovač vybere úlohu s nejkratší zbývajícím časem
- preemptivní**
- možnost vyhledovění dlouhých úloh
- příklad:**

Čas příchodu	Název úlohy	Doba úlohy (s)
0	P1	7
0	P2	5
3	P3	1



Multilevel feedback

- **N** různých úrovní – front (představují prioritu)
- **nepreemptivní**
- každá úroveň má svojí frontu úloh
- úloha vstoupí do systému do **nejvyšší fronty**
- na každé úrovni je stanoven maximální čas CPU, který může úloha obdržet
- **proces obsluhuje nevyšší neprázdnou frontu**



Plánování procesů v interaktivních systémech

- proces nesmí běžet příliš dlouho
- nevíme jak dlouhý bude CPU burst
- preemptivní**

Round Robin (RR)

- každému procesu přiřazene **časové kvantum**, po které může běžet
- běží-li proces na konci svého kvanta:**
 - preemce – spuštěn je další připravený proces
- proces skončí nebo se zablokuje před uplynutím kvanta:**
 - hned je spuštěn další připravený proces
- vhodná délka časového kvanta:**
 - krátké => snižuje efektivitu
 - dlouhé => zhoršuje dobu odpovědi
 - kompromis
 - nemusí být konstantné

RR + cyklické a prioritní plánování

- prioritní třídy** => v každé třídě procesy se stejnou prioritou
- prioritní plánování mezi třídami** => obsluhuje se třída s nejvyšší prioritou
- cyklická obsluha uvnitř třídy** => v rámci třídy se procesy cyklicky střídají



Plánovač spravedlivého sdílení

- **spravedlivé sdílení:** máme-li N uživatelů, každý dostne $1/N$ času
- **implementace:**
 - každý uživatel – položka g
 - obsluha přerušení časovače – inkrementuje g uživatele, kterému patří běžící proces
 - jednou za sekundu rozkla: $g = g/2$
 - priorita $P(p, g) = p - g$
 - pokud procesy uživatele využívaly CPU v poslední době – položka g je vysoká => vysoká penalizace

Plánování pomocí loterie

- **cíl:** poskytnou procesům příslušnou proporcí času CPU
- **princip:**
 - procesy obdrží tikety
 - plánovač vybere náhodně jeden tiket
 - vítězný proces obdrží cenu – 1 kvantum času CPU
 - důležitější procesy – více tiketů

Shrnutí

Algoritmus	Rozhodovací mód	Prioritní funkce	Rozhodovací pravidlo
RR	Preemptivní vyprší kvantum	$P() = 1$	cyklicky
prioritní	Preemptivní P jiný > P	Viz text	Náhodně, cyklicky
spravedlivé	Preemptivní P jiný > P	$P(p,g)=p-g$	cyklicky
loterie	Preemptivní vyprší kvant.	$P() = 1$	Dle výsledku loterie

Linux plánování

- **statická priorita:**
 - **základní**
 - pro RT úlohy (sched priority 1 až 99)
 - pro normální úlohy (sched priority 0)
 - **nice**
 - defaultně hodnota 0
 - lze nastavit -20 až 19
- **dynamická priorita:**
 - dynamická priorita = $\max(100, \min(\text{statická} - \text{bonus} + 5, 139))$
- **epocha:**
 - plánovač dělí čas na epochy
 - na začátku epochy dostane proces přidělený **time slice**
- **linux scheduler:**
 - procesy mají **time slice**
 - fronty:
 - 0 – 99 RT úlohy
 - 100 – 139 uživatelské úlohy
- **výpočet time slice:**
 - $SP < 120$: time slice = $(140 - SP) * 20$
 - $SP \geq 120$: time slice = $(140 - SP) * 5$

Priorita	SP	Nice	Time slice
Nejvyšší	100	-20	800 ms
Vysoká	110	-10	600 ms
Normální	120	0	100 ms
Nízká	130	+10	50 ms
Nejnižší	139	+19	5 ms

Completely Fair Scheduler (CFS)

- **red-black tree** místo front
 - klíč = vruntime (kolik času na CPU již proces spotřeboval)

- rovnoměrné rozdělení času procesům
- žadný idle procesor, pokud je co dělat
- **vruntime:**
 - říká jak moc si úloha zaslouží běžet
 - nižší číslo => zaslouží si běžet víc
- **priority:**
 - **decay factor** = jak rychle se zmenšuje čas pro běh tasku
- **princip:**
 - spočítá se čas strávený úlohou na CPU a vynásobí se koeficientem priority
 - přičte se do vruntime
 - pokud hodnota nemá nejmenší vruntime vystrídá jí úloha nejvíce nalevo v red-black tree
- **CFS:**
 1. naplánuj task s nejnižší vruntime
 2. task běží
 3. update vruntime
 4. znova vlož do stromu

Plánování v RT systémech

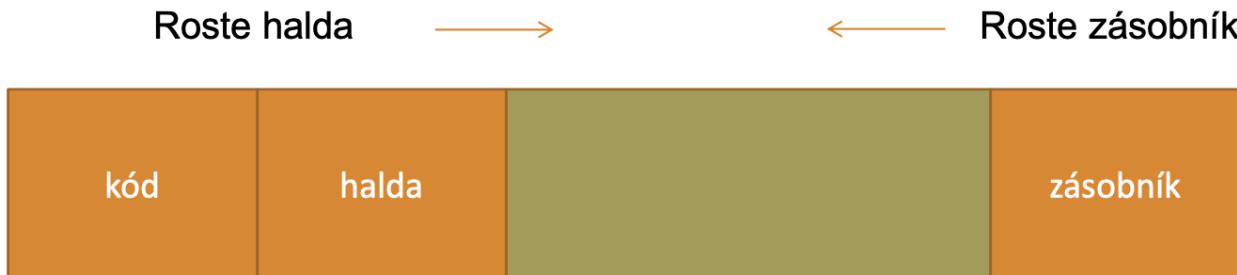
- RT proces reaguje na:
 - aperiodické události
 - periodické události
- **plánovatelné RT systémy:**
 - musí být možné všechny události včas zpracovat
 - zátěž lze zvládnout pokud platí:
 - $\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$
 - m – počet periodických událostí
 - P_i – perioda
 - C_i – sekundy

Dispatcher

- **dispatcher** = modul, který předá řízení CPU vybranému procesu short-term plánovačem
- **provede:**
 - přepnutí kontextu
 - přepnutí do uživatelského módu

- skok na danou instrukci v uživatelském procesu

Rozdělení paměti pro proces



Máme-li více vláken => více zásobníků, limit velikosti zásobníku



Synchronizace procesů

Časový souběh

- **časový souběh** = 2 nebo více procesů/vláken přistupují současně ke stejným zdrojům a výsledkem může být chyba
- **příklad:** 2 procesy zvětšují asynchronně sdílenou proměnnou X

Řešení

- čtení a modifikace atomicky
- HW řešení
- SW řešení – v 1 omažík může číst a zapisovat společná data pouze 1 proces

Pravidla řešení

1. **vzájemné vyloučení** = žádné 2 procesy nesmí být současně uvnitř KS

2. proces mimo KS nesmí blokovat jiné procesy
3. žádný proces nesmí čekat nekonečně dlouho pro vstup do KS

Možnosti řešení

1. zákaz přerušení
2. aktivní čekání
3. zablokování procesu

Zákaz přerušení

- nedochází k přeplánování procesů

zakaž přerušení;
kriticiká sekce;
povol přerušení;

- nejjednodušší řešení

Aktivní čekání

- **předpoklady:**
 - zápis a čtení ze společné datové oblasti jsou **atomické** operace
 - KS nemohou mít přiřazeny prioritu
 - relativní rychlosť procesů je neznámá
 - proces se může pozastavit mimo KS
- **princip:**
 - průběžné testování proměnné ve smyčce dokud nenabude očekávanou hodnotu
- **nevýhody:**
 - plýtvá časem CPU

Peterson

```
void enter_CS(int process) {  
    int other;  
    other = 1 - process;  
    interested[process] = true;  
    turn = process;  
    while ((turn == process) && (interested[other]) = true);  
}
```

```
void leave_CS(int process) {
    interested[process] = false;
}
```

Spin lock s instrukcí TSL

- **TSL** = CPU instrukce, která otestuje hodnotu a nastaví paměťové místo v jedné atomické operaci
- TSL R, lock udělá najednou:
 - LD R, lock
 - LD lock, 1
- **implementace:**

```
Spin_lock:
    TSL R, lock
    CMP R, 0
    JNE spin_lock
    RET
Spin_unlock:
    LD lock, 0
    RET
```

Zablokování procesu

- využívají se semafory

Semafory

- **semafor** = proměnná, obsahuje nezáporné celé číslo
- operace P(s) a V(s)
- **struktura:**

```
typedef struct {
    int hodnota;
    process_queue *fronta;
} semaphore;
```

- **operace P(S):**

```
void P(semaphore S) {
    if (S.hodnota > 0) {
```

```

        S.hodnota -= 1;
    } else {
        S.process_queue.add(process);
    }
}

```

- **oprace V(S):**

```

void P(semaphore S) {
    if (!S.procces.queue.empty()) {
        S.process_queue.deque() // vzbud' proces
    } else {
        S.hodnota += 1;
    }
}

```

Kritická sekce

- **kritická sekce** = místo v programu, kde dochází k přístupu ke společným datům

Vzájemné vyloučení – semafory

```

var s: semaphore = 1;
cobegin
    while true do:
        begin
            P(s);
            KS1;
            V(s);
        end
    ||
    while true do:
        begin
            P(s);
            KS1;
            V(s)
        end
coend

```

Producent – konzument

- základní synchronizační úloha
- **cíl:** synchronizovat přístup ke sdílenému bufferu
- **řešení pomocí semaforů:**

```

semaphore e = N;
semaphore f = 0;
semaphore m = 1;

cobegin
    begin while true do { producent }
        produkuj;
        P(e);
        P(m);
        vloz;
        V(m);
        V(f)
    end
    ||
    begin while true do { konzument }
        P(f);
        P(m);
        vyber;
        V(e)
        zpracuj;
    end
coend

```

Mutexy

- **mutex** = mutual exclusion
- synchronizační prvek pro vzájemné vyloučené
- **mutex VS semafory:**
 - **mutex** => stejné vlákno zamyká i odemyká
 - **semafor** => jedno vlákno může zamknout a jiné odemknout
- využívají se při implementaci semaforů pro zajištění atomičnosti operací P a V

Implementace pomocí TSL

```
mutex_lock:  
    TSL R, mutex
```

```
CMP R, 0  
JE ok  
CALL yield
```

```
JMP mutex_lock  
ok: RET
```

```
mutex:unlock:  
    LD mutex, 0  
    RET
```

- **operace yield** = volající se dobrovolně vzdá procesoru ve prospěch jiných vláken
 - jádro OS proces volající yield přeplánuje

Minotory

- jazyková konstrukce
- v jednu chvíli může být aktivní pouze jeden proces uvnitř monitoru
- blok podobný proceduře nebo funkci
- **uvnitř definované:**
 - promenné
 - procedury
 - funkce
- **promenné monitoru:**
 - neviditelné z venčí
 - dostupné pouze funkcím monitoru
- **funkce monitoru:**
 - viditelné a volatelné vně monitoru

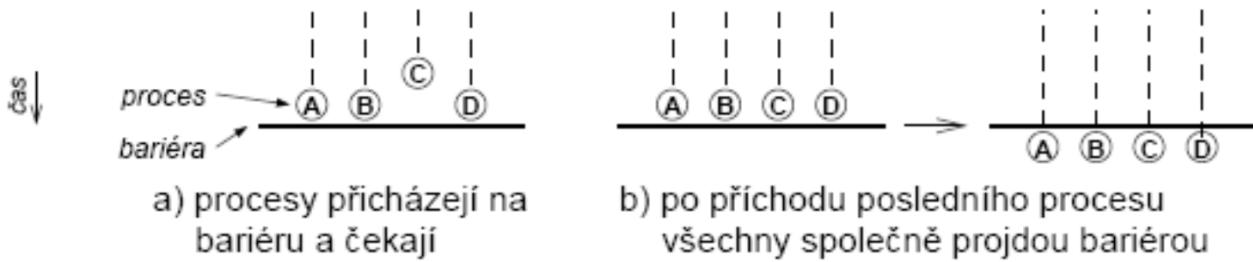
Podmínkové proměnné

- **podmínkové proměnné** = speciální typ proměnné
- představují formu procesů, které na danou podmítku čekají
- **operace:**
 - `wait(C)`

- volající pozastaven nad podmínkou C
- pokud je nějaký proces připraven vstoupit do minotoru, bude mu to dovoleno
- **signal(C)**
 - existuje-li 1 či více procesů pozastavených nad podmínkou C je jeden z nich reaktivován
 - **problém:** mohli by běžet dva procesy v monitoru
 - **řešení:**
 - **Hoare** = proces volající `signal(C)` se pozastaví
 - **Hansen** = `signal(C)` lze volat pouze jako poslední příkaz v monitoru
 - **Java** = čekající proces může začít běžet až po té co proces volající `signal(C)` opustí monitor

Bariéry

- synchronizace skupiny procesů



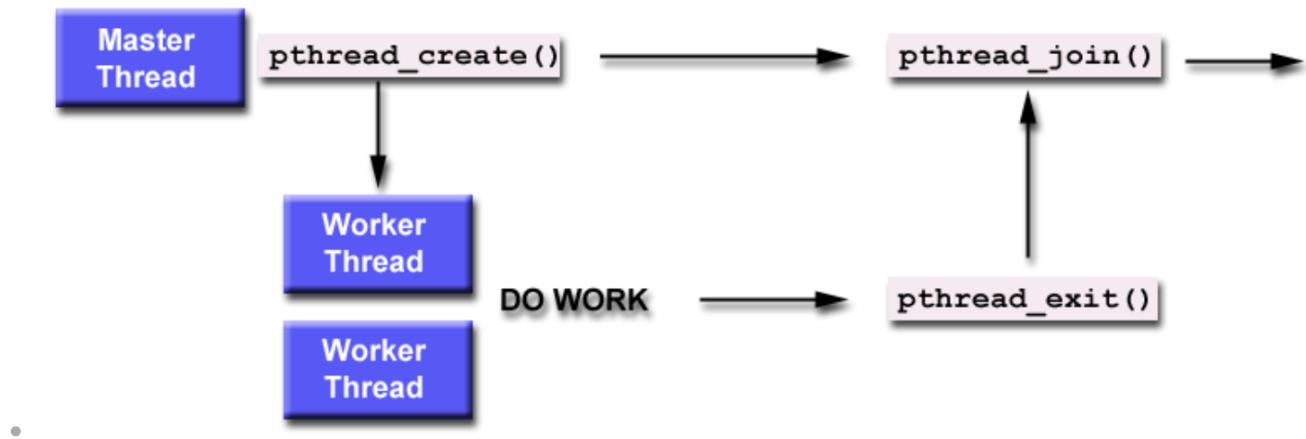
Vlánka

- **sdílejí:**
 - adresní prostor
 - otevřené soubory
- **mají skoukromý:**
 - PC
 - obsah registrů
 - zásobník
 - lokální proměnné
 - plánovací vlastnosti

Multithreading

- více vláken pracuje současně
- proces začíná běh s jedním vláknem, další vytváří za běhu
- menší režie než více procesů

Čekání na dokončení vláken



Meziprocesová komunikace – IPC

- **IPC** = Interprocess Communication
- **možnosti komunikace:**
 - sdílená paměť
 - zasílání zpráv

Linux – signály

- **signál** = speciální zpráva zaslána jádrem OS procesu (OS → proces)
- inicializátor může být proces
- zpráva obsahuje jen číslo signálu
- **asynchronní**
- např. SIGTERM, SIGKILL

příkaz	popis
ps aux	Informace o procesech
kill -9 1234	Pošle signál č. 9 (KILL) procesu s PID číslem 1234
man kill	Návod k příkazu kill
man 2 kill	Návod k volání kill
man 7 signal	Návod k signálům
kill -l	Vypíše seznam signálů

- **události generující signály:**

- stisk kláves
- příkaz kil
- programy

- **reakce na signály:**

- standardní zpracování:
 - ukončí proces (Term)
 - zastaví (Stop), pustí zastavený (Cont)
 - ukončí a provede dump (Core)
- vlastní zpracování funkcí
- ignorování signálu

Datové roury

- **datové roury** – jednosměrná komunikace mezi 2 procesy
- data zapisovaná do roury jedním procesem lze dalším hned číst
- např. `cat /etc/passwd | grep josef | wc -l`

Příklad použití roury

```
int main() {
    int pipefd[2];
    pipe(pipefd);
    if (fork() == 0) {
        // potomek čte z roury
        close(pipefd[1]) // zavřeme výstup
```

```

        read(pipedf[0], ...); // čteme ze vstupu
    } else {
        // rodič do roury zapisuje
        close(pipefd[0]); // zavřeme vstup
        write(pipefd[1], ...); // zapisujeme na výstup
    }
}

```

Předávání zpráv – send, receive

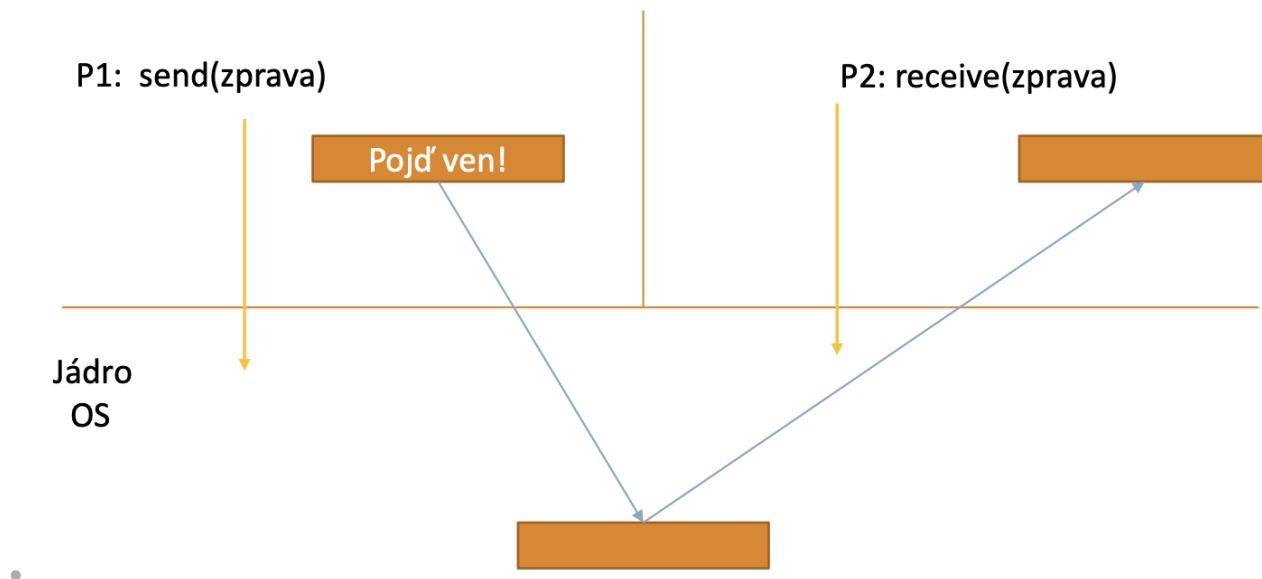
- **2 primitiva:**

- send(adresát, zpráva)
 - zpráva = libovolný datový objekt
- receive(odesílatel, zpráva)
 - přijatá zpráva se uloží do proměnné

- **vlastnosti:**

- synchronizace
- unicast, multicast, broadcast
- přímá X nepřímá komunikace
- délka fronty zpráv
- pevné X proměnná délka zprávy

Volání jádra



Synchronizace

- **blokující (synchronní):**

- volání se zablokuje, dokud požadovaná událost nenastane
- `receive()` – čeká na zprávu
- **neblokující (asynchronní):**
 - volání hned pokračuje
 - `send()` – ůředá zprávu jádru a dál se nastará

Varianty

- **blokující send** = čeká na převzetí zprávy příjemcem
- **neblokující send** = vrací se ihned po odeslání zprávy
- **blokující receive** = není-li ve frontě žádná zpráva zablokuje se

Receive s omezeným čekáním

- `receive(odesílatel, zpráva, t)`
 - čeká na příchod zprávy dobu `t`
 - pokud zpráva nepřijde do doby `t` vrací se volání s chybou

Adresování

- **multicast** = zprávu pošleme skupině procesů
- **broadcast** = zprávu pošleme všem procesům

Délka fronty zpráv (buffering)

- **nulová délka** = žádná zpráva nemůže čeka
 - odesílatel se zablokuje a čeká na příjemce – **rendezvous**
- **omezená kapacita** = blokování při dosažení kapacity
- **neomezená kapacita** = odesílat se nikdy nezablokuje

Linux – systémové volání

- `msgsnd()` – pošle zprávu (konkrétní implementace `send()`)
- `msgrcv()` – přijme zprávu z fronty zpráv (konkrétní implementace `receive()`)

Producent–konzument pomocí zpráv

- **předpoklady:**
 - `send()` – neblokující
 - `receive()` – blokující, příjem od libovolného adresáte
 - fronta zpráv – dostatečně velká

- zprávy doručeny v pořadí FIFO a neztrácejí se
- symsetrický problém
- producent generuje plné položky
- konzument generuje prázdné položky

```

cobegin
    begin
        while tru do
            produkuj;
            receive(konzument, m); // blokující operace
            m = zaznam;
            send(konzument, m);
    end
    ||
begin
    for i=1 to N do
        send(producent, e);
    while true do
        recieve(producent, m); // blokující operace
        zaznam = m;
        send(producent, e);
        zpracuj;
    end
coend

```

Adresování fronty zpráv

- proces pošle zprávu => zpráva se připojí k určené frontě zpráv
- jiný proces přijme zprávu => vyjmě zprávu z dané fronty zpráv

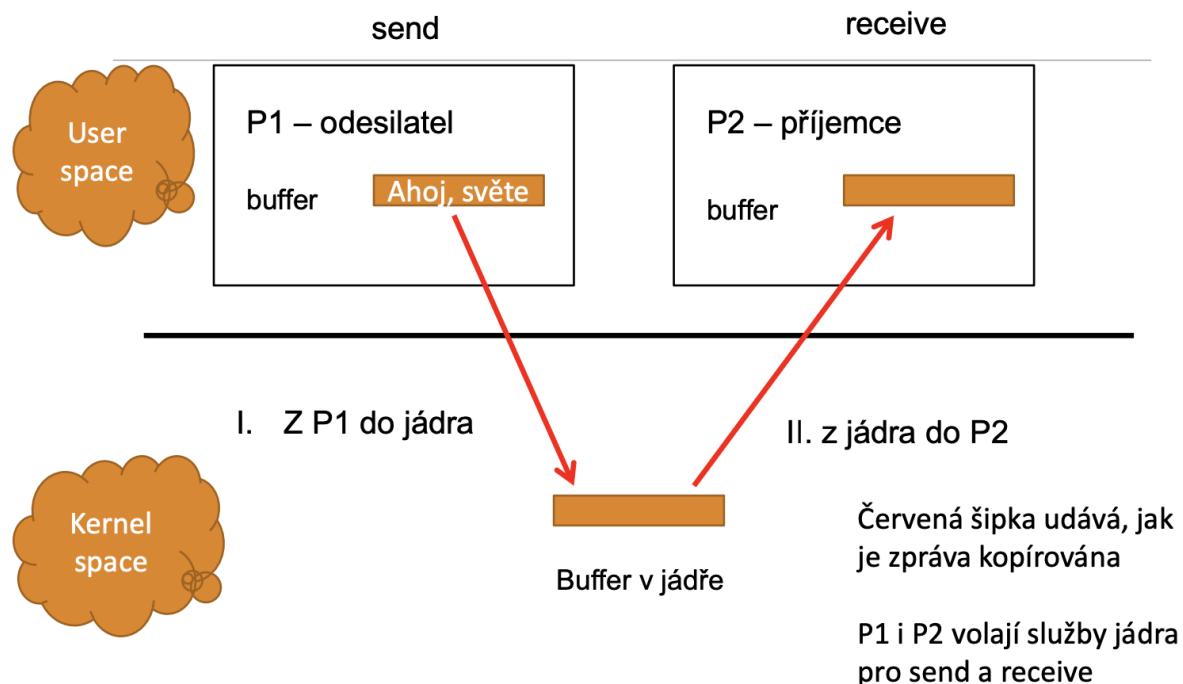
Mailbox, port

- **mailbox** = fronta zpráv využívaná více odesílateli a příjemci
 - obecné shcéma
 - `receive()` – drahé
- **port** = omezená forma mailboxu
 - zprávy může vybírat pouze jeden příjemce



Lokální komunikace

- na stejném stroji – snížení režie zpráv
- dvojí kopírování:**
 - z procesu odesílatele do fronty v jádře
 - z jádra do procesu příjemce



- rendezvous:**
 - eliminuje frontu zpráv v jádře (1 kopírování)
 - až jsou obě volání OS, send i receive – zprávu zkopirovat z odesílatele přímo do příjemce
- využítí mechanismu **virtuální paměti**:
 - paměť obsahující zprávu přemapována
 - z procesu odesílatele
 - do procesu příjemce
 - zpráva se nekopíruje

RPC (Remote Procedure Call)

- používání `send`, `receive`

- dovolit procesům (klientům) volat procedury umístěné v jiném procesu (serveru)

- **local X remote call:**

Local call:

```
int add (int a, int b) {
    int result;

    result = a + b;

    return result;
}
```

Remote (RPC) call:

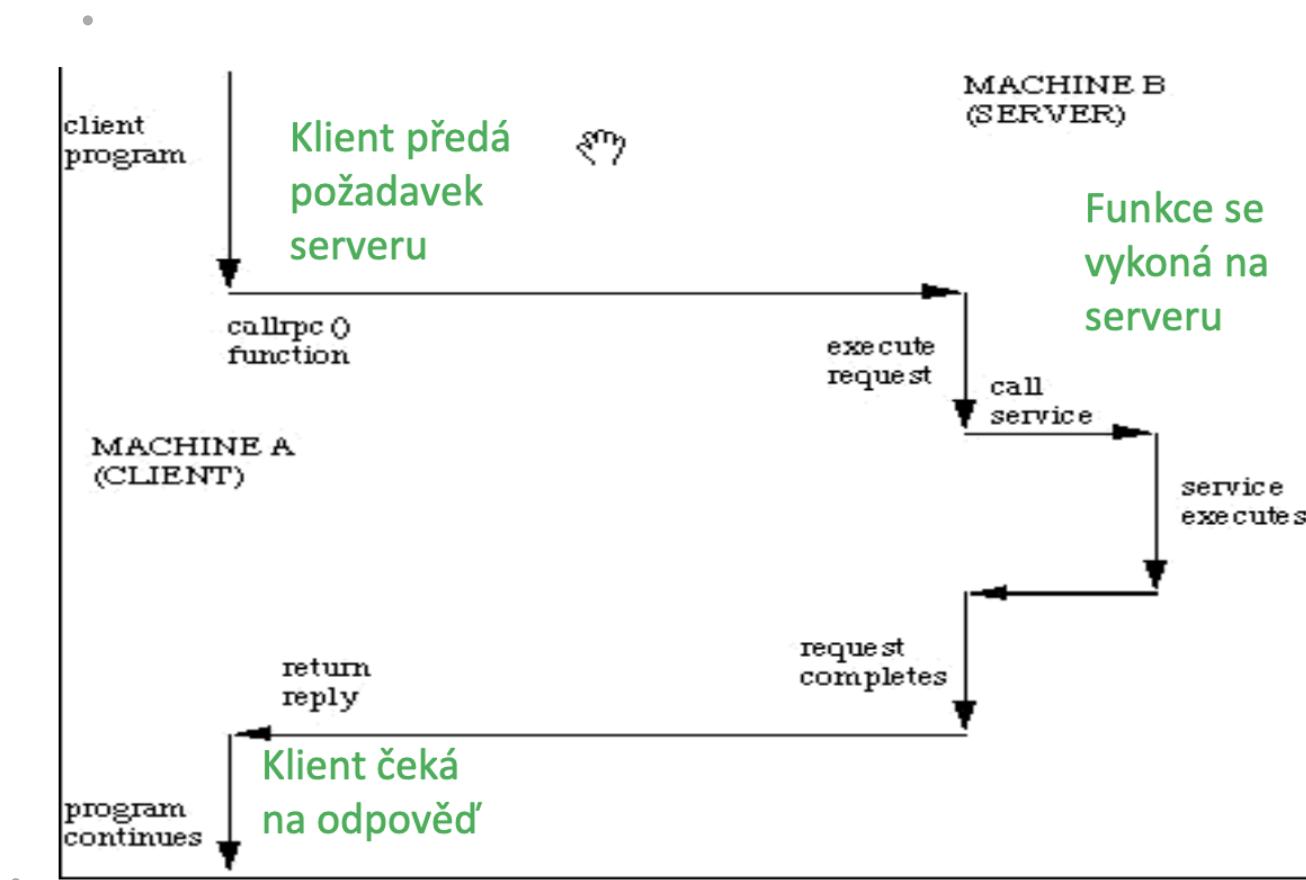
```
int add (int a, int b) {
    int result;

    send ("server, call function add", a, b);

    /* server provides the computation */

    receive (result);

    return result;
}
```



Spojka klienta, serveru

- klientský program sestaven s knihovní funkcí = spojka klienta (client stub)
- program serveru sestaven se spojkou severu (server stub)

- spojky zakrývají, že volání není lokální
- **kroky komunikace:**



-

 1. klient zavolá spojku, reprezentující vzdálenou proceduru
 2. spojka klienta zabalí argumenty do zprávy a pošle jí serveru
 3. spojka serveru zprávu přijme, vezme argumenty a zavolá proceduru
 4. procedura se vrátí, návratová hodnota se pošle spojkou serveru klientovi
 5. spojka klienta přijme zprávu obsahující návratovou hodnotu a předá jí volajícímu

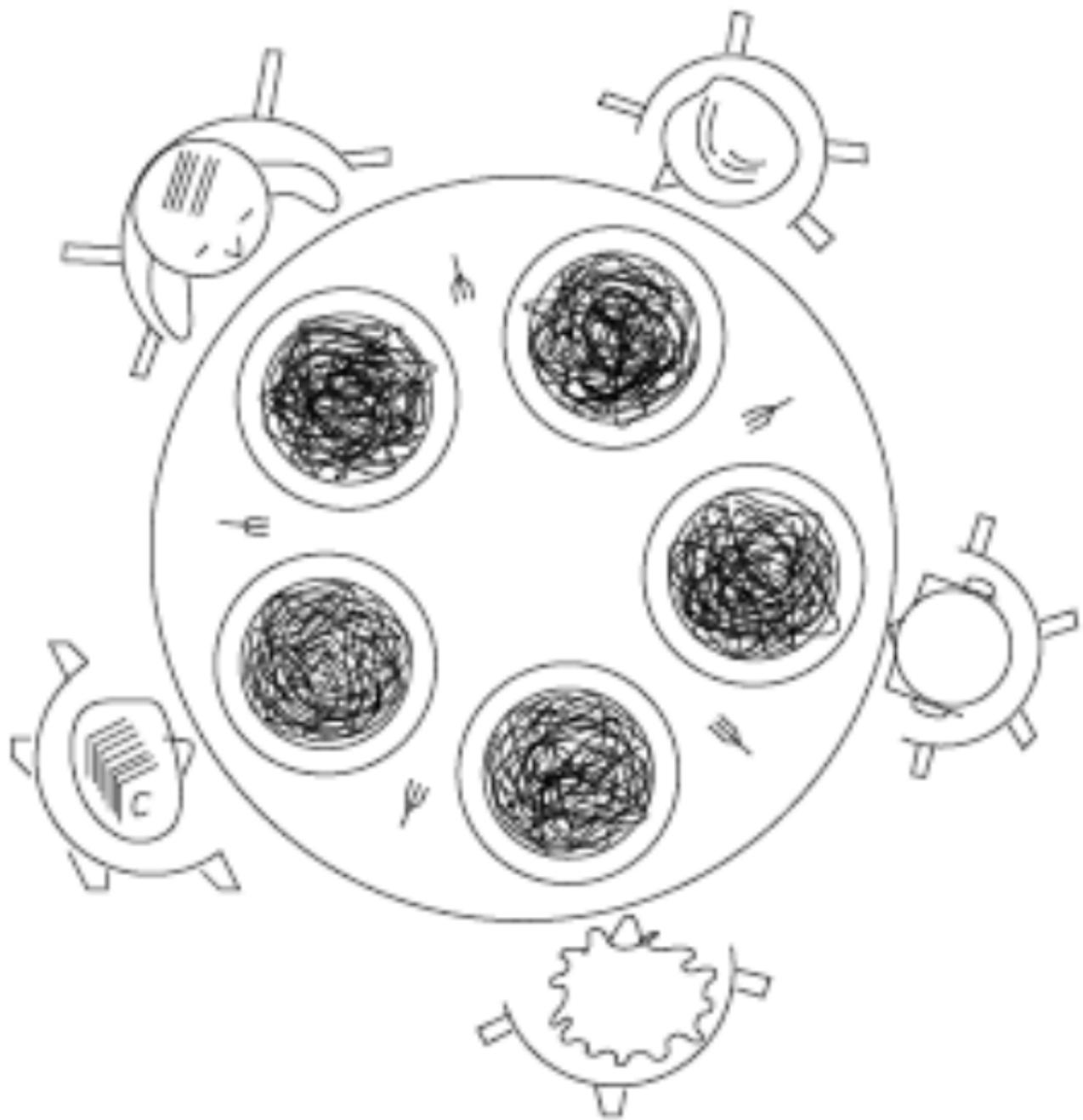
Reprezentace informace

- stroje mohou mít různé architektury:
 - rozdílná vnitřní reprezentace datových typů
 - rozdílné kódování řetězců
 - rozdílné numerické typy
- **big endian:**
 - nejvýznamnejší byte (MSB) na nejnižší adrese
 - v paměti od nejnižší adresy: 4a, 3b, 2c, 1d
- **little endian:**
 - nejméně významný byte (LSB) na nejnižší adrese
 - v paměti od nejnižší adresy: 1d, 2c, 3b, 4a

Klasické problémy

- Producent–konzument
- Večeřící filozofové
- Čtenáři písáři
- Spící holič

Problém večeřících filozofů



- filozof potřebuje 2 vidličky aby mohl jíst

Problém uvíznutí (deadlock)

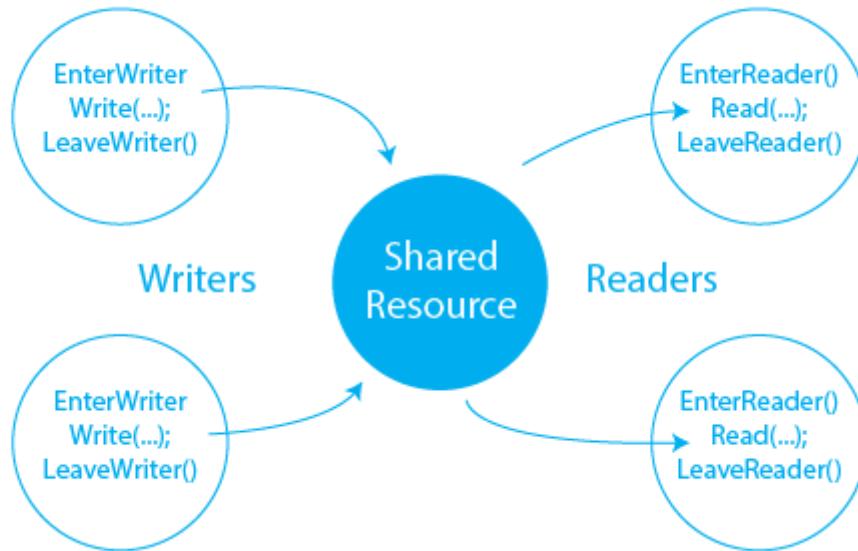
- **deadlock** = cyklické čekání 2 a více procesů na událost, kterou může vyvolat pouze jeden z nich ale nikdy k tomu nedoje
- **analogie filozofů:** všichni zvednou najednou levou vidličku

Problém vyhlovění (starvation)

- **vyhlovění** = proces se nikdy nedostane k požadovaným zdrojům
- **analogie filozofů:** filozof nikdy nebude mít 2 vidličky

Problém čtenářů a písářů

- modeluje přístup do DB
- čtenáři chtějí číst
- písáři chtějí zapisovat



•

Deadlock

- **deadlock** = cyklické čekání 2 a více procesů na událost, kterou může vyvolat pouze jeden z nich ale nikdy k tomu nedojde
- **zdroj** = označuje zařízení, záznam a jiné
 - **přeplánovatelný** = lze je odebrat procesu bez škodlivých efektů
 - **nepřeplánovatelný** = proces zhavaruje pokud jsou mu odebrány
 - **sériově využitelné zdroje** = proces zdroj alokuje, používá, uvolní
 - **konzumovatelné zdroj** = například zprávy, které produkuje jiný proces
- **práce se zdrojem:**
 - **žádost** – uspokojena bezprostředně nebo proces čeká (systémové volání)
 - **použití** – například tisk na tiskárně
 - **uvolnění** – proces uvolní zdroj (systémové volání)

Coffmanovi podmínky (podmínky vzniku deadlocku)

1. vzájemné vyloučení

- každý zdroj je buď dostupný nebo je výhradně přiřazen jednomu procesu

2. hold and wait

- proces držící výhradně přiřazené zdroje může požadovat další zdroje

3. nemožnost odejmoutí

- jednou přiřazené zdroje nemohou být procesu násilně odejmuty

4. cyklické čekání

- musí být cyklický řetězec 2 nebo více procesů, kde každý z nich čeká na zdroj držený dalším členem

Modelování deadlocku

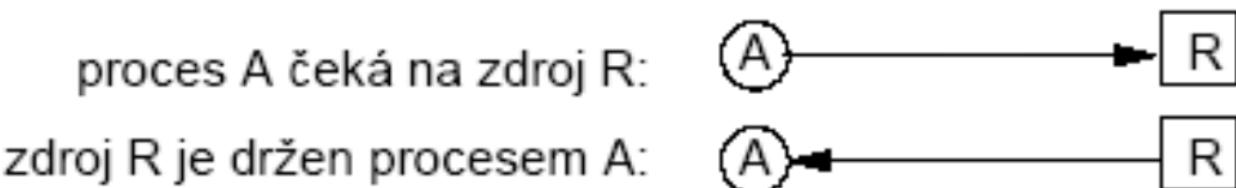
- **graf alokace zdrojů**

- **2 typy uzelů:**

- **proces** = kruh
- **zdroj** = čtverec

- **hrany:**

- hrana zdroj → proces = zdroj držen procesem
- hrana proces → zdroj = proces blokován čekáním na zdroj



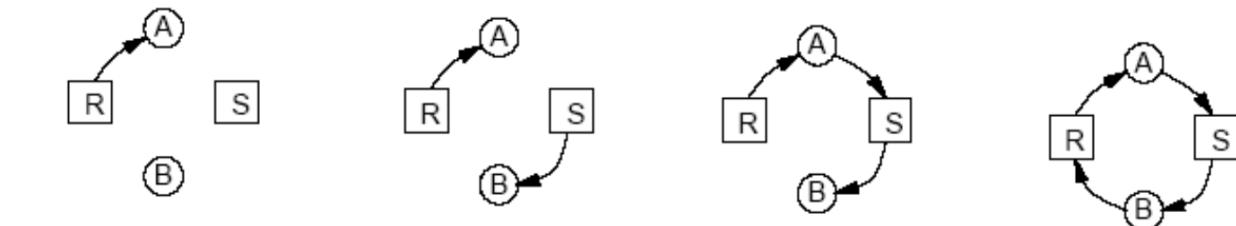
•

- **cyklus v grafu => nastalo uvíznutí** – nutná a postačující podmínka

- **příklad:**

- zdroje: Rekorder R a Scanner S
- proces: A, B

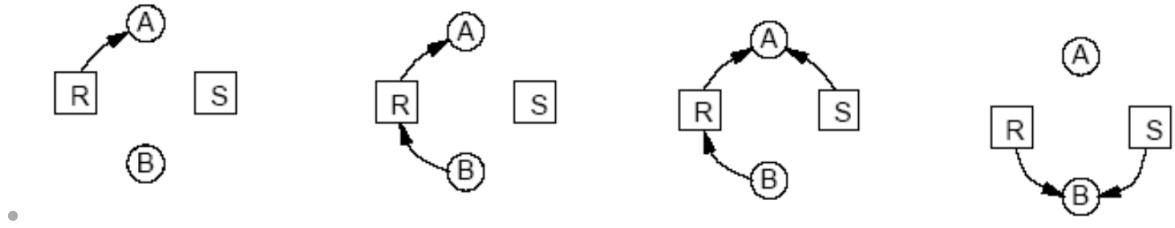
1. A žádá R dostane, B žádá S dostane
2. A žádá S a čeká, B žádá R a čeká => uvíznutí



•

- **pořadí alokace:**

- pokud A, B žádají o zdroje R a S ve stejném pořadí uvíznutí nenastane
1. A žádá R a dostane, B žádá R a čeká
 2. A žádá S a dostane, A uvolní R a S
 3. B čekal na R a dostane, B žádá S a dostane



Vypořádání s deadlockem

1. ignorovat problém
2. detekce a zotavení
3. dynamické zabránění pomocí pečlivé alokace zdrojů
4. prevence, pomocí strukturální negace jedné z dříve uvedených nutných podmínek pro vznik uvíznutí

1. Ignorování problému

- předstíráme, že problém neexistuje
- neřešíme protože je vysoká cena za eliminaci deadlocku
- žádný ze známých OS se nezabývá uvíznutím uživatelských procesů

2. Detekce a zotavení

- systém se nesnaží zabránit vzniku deadlocku
- detekuje deadlock
- pokud nastane deadlock, provede akci pro zotavení
- detekování cyklu v grafu
- **zotavení pomocí preempce:**
 - vlastníkovi zdroj dočasně odejmout
 - závisí na typu zdroje – často obtížné či nemožné
- **zotavení pomocí zrušení změn (rollback)**
 - častá uvíznutí – checkpointing procesů = zápis stavu procesů do souboru, aby proces mohl být v případě potřeby vrácen do uloženého stavu
 - detekce uvíznutí – nastavení na dřívější checkpoint, kde proces ještě zdroje nevlastnil
- **zotavení pomocí zrušení procesu:**
 - nejhorší způsob – zrušíme jeden nebo více procesů
 - zrušení procesu v cyklu

3. Dynamické zabránění

- procesy žádají o zdroje po jednom
- systém rozhodně, zda je přiřazení zdroje bezpečné, nebo nehrozí deadlock
- pokud je bezpečné – zdroj přiřadí, jinak pozastaví žádající proces
- stav je bezpečný pokud existuje alespoň jedna posloupnost, ve které mohou procesy doběhnout bez uvíznutí
- **bankéřův algoritmus pro jeden typ zdroje:**
 - bankéř (procesor) na malém městě, 4 zákazníci (procesy) – A, B, C, D
 - každému garantuje půjčku (6, 5, 4, 7) = 22 dohromady
 - bankéř ví že všichni nebudou chtít půjčku najednou, pro obsluhu si nechává pouze 10

Zákazník (proces)	Má půjčeno	Max. půjčka
A	0	6
B	0	5
C	0	4
D	0	7

dostupné zdroje: 10

stav: SAFE

Zákazník (proces)	Má půjčeno	Max. půjčka
A	1	6
B	0	5
C	0	4
D	0	7

dostupné zdroje: 9

stav: SAFE

Zákazník (proces)	Má půjčeno	Max. půjčka
A	1	6
B	1	5
C	0	4
D	0	7

dostupné zdroje: 8

stav: SAFE

Zákazník (proces)	Má půjčeno	Max. půjčka
A	1	6
B	1	5
C	2	4
D	0	7

dostupné zdroje: 6

stav: SAFE

Zákazník (proces)	Má půjčeno	Max. půjčka
A	1	6
B	1	5
C	2	4
D	4	7

dostupné zdroje: 2

stav: SAFE

Jediná validní půjčka v tomto stavu je zákazníkovi C

Zákazník (proces)	Má půjčeno	Max. půjčka
A	1	6
B	1	5

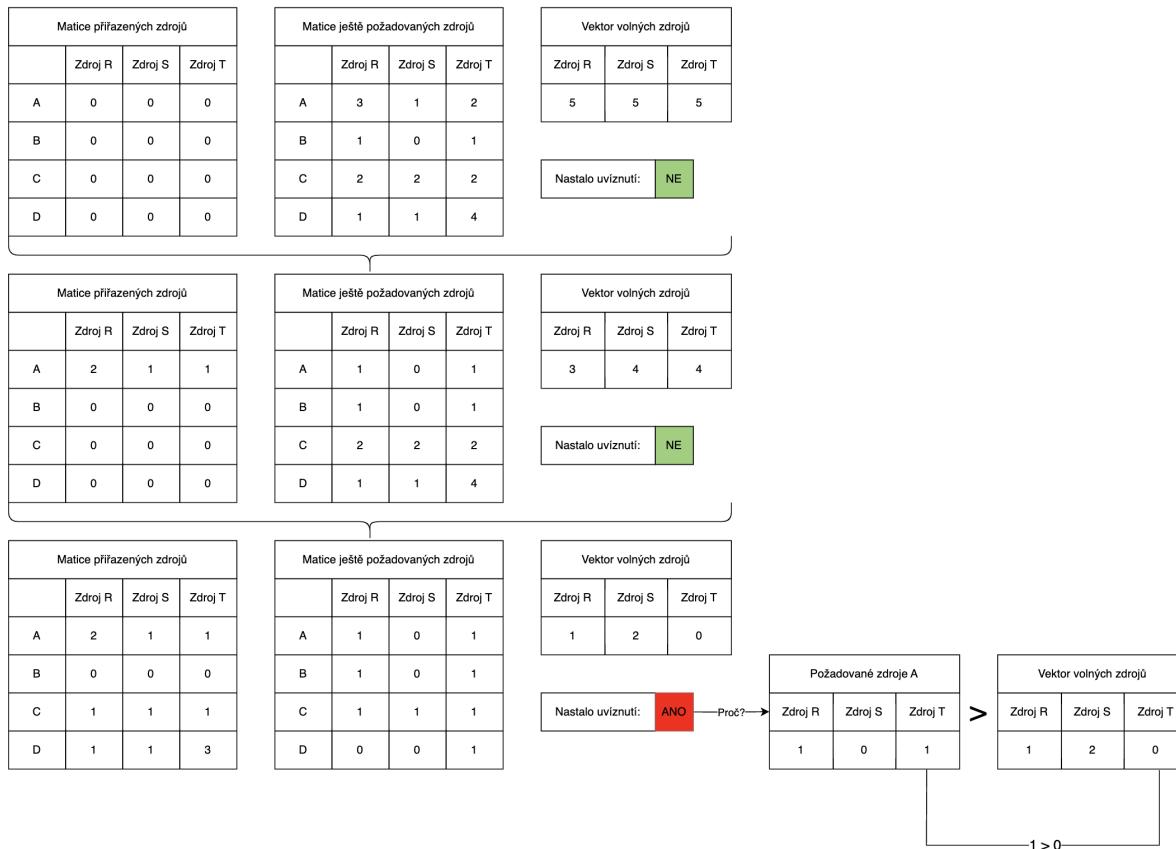
dostupné zdroje: 1

stav: NOT SAFE

Nemůžete nikomu poskytnout



- **bankéřův algoritmus pro N typů zdroje:**



4. Prevence uvíznutí

- snaží se předcházet naplnění Coffmanovo podmínek
- negujeme tyto podmínky
- **P1 – vzájemné vyloučení:**
 - **prevence** – zdroj nikdy nepřiřadit výhradně
 - problém lze řešit jen pro některé zdroje (tiskárna)
 - **spooling** = dočasné ukládání dat z tiskárny na disk
- **P2 – hold and wait:**
 - **prevence** – požadovat, aby procesy alokovaly všechny zdroje před svým spuštěním
- **P3 – nemožnost zdroje odejmout:**
 - odejímat zdroje je velmi obtížné
 - proces může nechat zdroj v nekonzistentním stavu
- **P4 – cyklické čekání**
 - proces může mít jediný zdroj, pokud chce jiný musí předchozí uvolnit

- očíslovat zdroje a požadavky provádět v číselném pořadí

Shrnutí

1. **ignorování problému** = většina OS ignoruje uvíznutí uživatelských procesů
2. **detekce a zotavení** = pokud uvíznutí nastane detekujeme a něco s tím uděláme (rollback, zrušení proces)
3. **dynamické zabránění** = zdroj přiřadíme pouze pokud bude stav bezpečný (bankéřův algoritmus)
4. **prevence** = strukturálně negujeme jednu z Coffmanovo podmínek
 - **vzájemné vyloučení** – spooling všeho
 - **hold and wait** – procesy požadují zdroje na začátku
 - **nemožnost odejmoutí** – odejmi (nefunguje)
 - **cyklické čekání** – zdroje očislujeme a žádáme v číselném pořadí

Vyhledování

- **vyhledování** = proces zdroj nikdy neobdrží
- například u strategie SJF (Shortest Job First)

Správa souborů

Základní znalosti

- v PC může být více HDD
- každý disk se může dělit na oddíly:
 - `/dev/sda1, /dev/sda2, /dev/sda3` (1. disk má 3 oddíly)
 - `/dev/sdb1` (2. disk má 1 oddíl)
- každý oddíl má svůj FS
- `fdisk /dev/sda` = rozdělení disku na oddíly
- `/sbin/mkfs.ext4 /dev/sda1` = formátování oddílu na vybraný FS

Pevný disk

cluster, sektor

- **cluster** = nejmenší alokovatelná jednotka skládající se ze sektorů
- **sektor** = základní adresovatelná fyzická jednotka na disku

LBA vs cluster

- **cluster** = skupina 1 a více sektorů
- **LBA** = jeden sektor na disku

CHS a LBA adresování

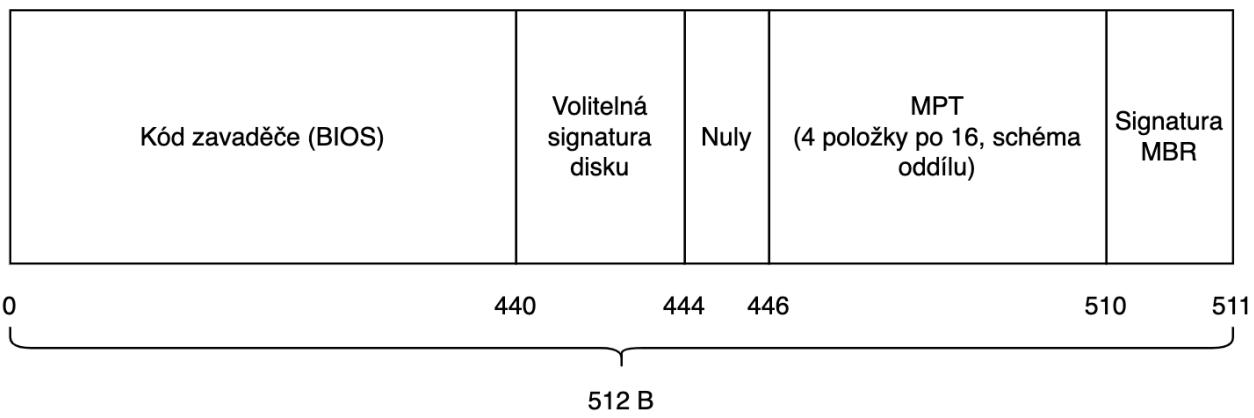
- **CHS (Cylinder-Head-Sector, Stopa-Hlava-Sektor):**
 - uvedeme stopu, hlavu disku, číslo sektoru na stopě
- **LBA (Logical Block Adressing)**
 - čísluje sektory na disku lineárně

Dělení disku na oddíly

Master Partition Table (MPT)

- **Master Boot Record (MBR)** – na počátku disku
- umožňuje 4 primární oddíly
 - **chceme-li víc:**
 - uděláme 3 primární a 1 extended
 - extended jde dělit na další

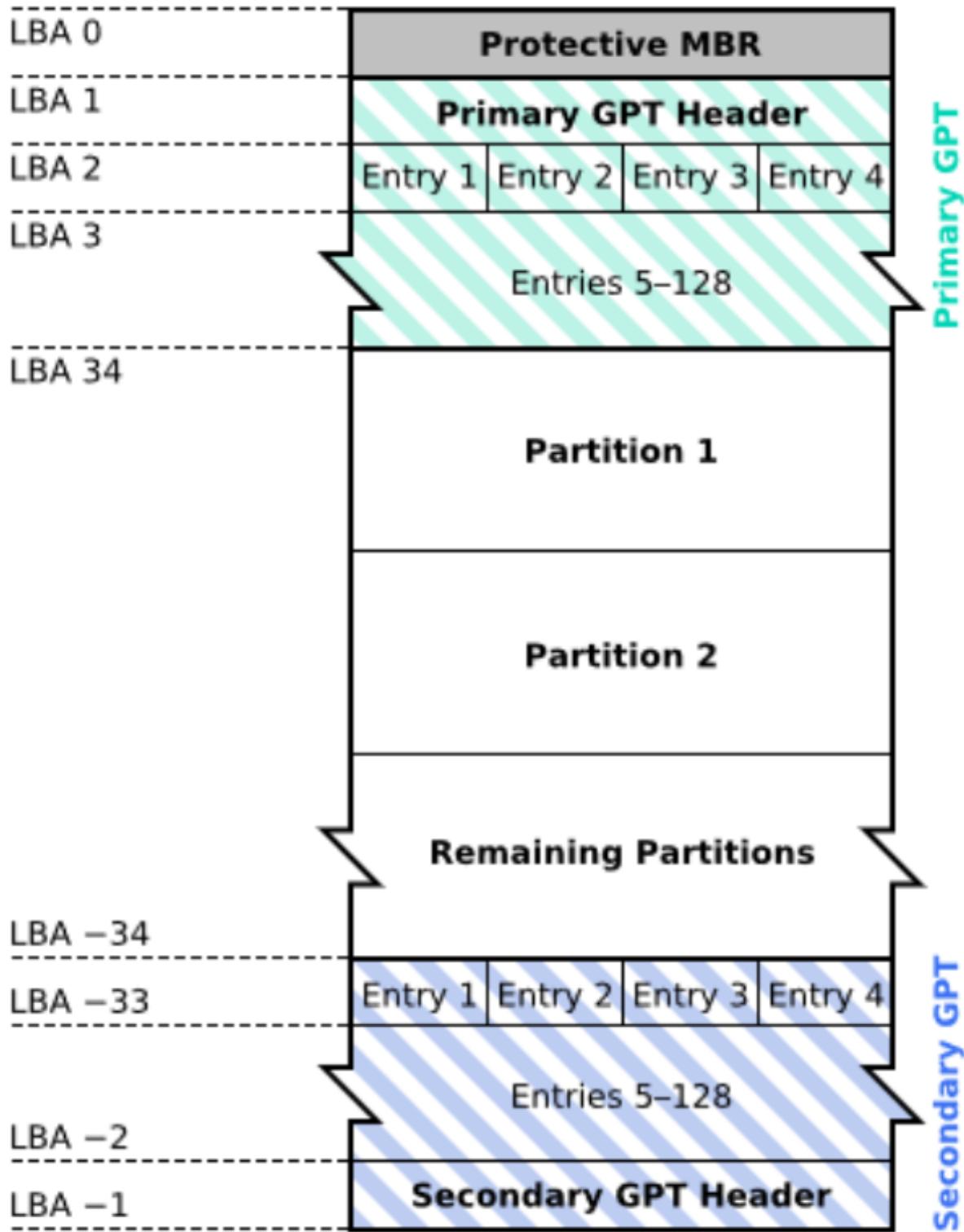
Struktura MBR



GUID Partition Table (GPT)

- nelimituje na 4 oddíly
- **používá:** Mac OS, Windows

Struktura GPT



- záložní kopie uložena na konci disku
- velikost = 512B LBA = 34x512B = 16KB
- **LBA (Logical Block Addressing)** = logický blok
 - lineární číslování bloků od 0

Soubor

-
- **soubor** = pojmenovaná sekvence bytů
 - OS poskytuje abstrakci FS, aby programy mohli pracovat stejně s různými FS

Typy souborů

- **obyčejné soubory:**
 - data zapsaná aplikacemi
 - textové nebo binární
- **adresáře:**
 - udržují strukturu FS
- **speciální:**
 - znakové soubory (/dev/tty)
 - blokové soubory (/dev/sda)
 - pojmenované roury
 - symbolické odkazy

Vnitřní struktura (obyčejného) souboru

- **nestrukturovaná posloupnost bytů** = OS obsah nezajímá, interpretuje aplikace
- **posloupnost záznamů** = každý záznam má vnitřní strukturu
- **strom záznamů** = záznamy řazeny podle klíče

Způsob přístupu k souboru

- **sekvenční:**
 - čtení dat v pořadí v jakém jsou uloženy
 - `rewind` = čtení od začátku
- **přímý (random access):**
 - čtení v libovolném pořadí nebo podle klíče
 - `lseek` = pozice čtení/zápisu

Atributy

- **atribut** = info sdružené se souborem
- **příznaky** = vlastnosti souboru
 - **hidden** = neobjeví se při výpisu
 - **archive** = nezálohovaný soubor

- **temporary** = automaticky zrušen
- **read-only**

Služby pro práci se soubory

služba	popis
create	vytvoří soubor
open	otevře soubor
read	čtení
write	zápis
Iseek	změna pozice
close	uzavření souboru

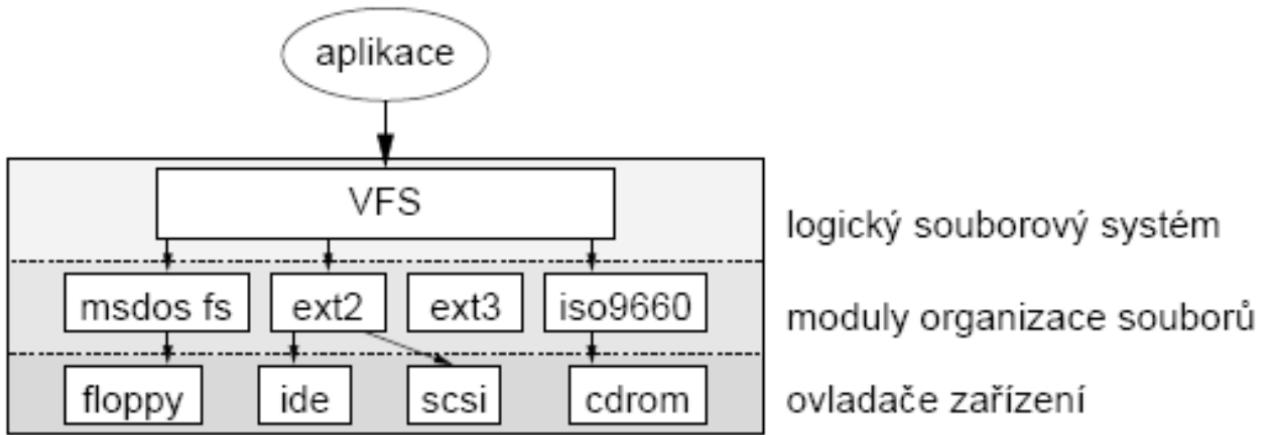
Paměťově mapované soubory

- mapování souborů do adresního prostoru procesu
- `mmap()`, `munmap()`

Souborové systémy

- **souborový systém** = způsob organizace dat ve formě souborů (a adresářů), tak aby k nim bylo možné snadno přistupovat
- **důvod:** potřeba aplikací trvale uchovávat data
- **požadavky:**
 - ukládání velkého množství dat
 - strukturovanost
 - informace zachována i po ukončení procesu
 - data přístupná více procesům
- **problémy při přístupu k zařízení:**
 - alokace prostoru na disku
 - pojmenování dat
 - přístupová práva
 - zotavení po havárii

Implementace



- **VFS** – volán aplikacemi
 - převádí jméno souboru na informaci o souboru
 - udržuje informaci o otevřeném souboru
 - ochrana a bezpečnost
- **modul organizace souborů** – konkrétní FS
 - čte/zapisuje datové bloky
 - správa volného prostoru + alokace volných bloků
- **ovladače zařízení** – pracuje s daným zařízením
 - interpretují požadavky

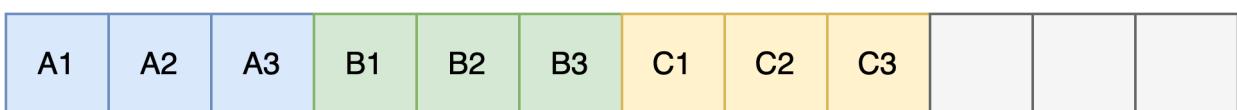
Komunikace VFS s konkrétním FS

- nový FS se zaregistrouje
- díky registraci VFS ví jak volat jeho metody `open`, `read`, `write`
- při požadavku na soubor VFS napřed zjistí, na kterém FS leží

Způsoby alokace

Kontinuální alokace

- kontinuální posloupnost diskových bloků

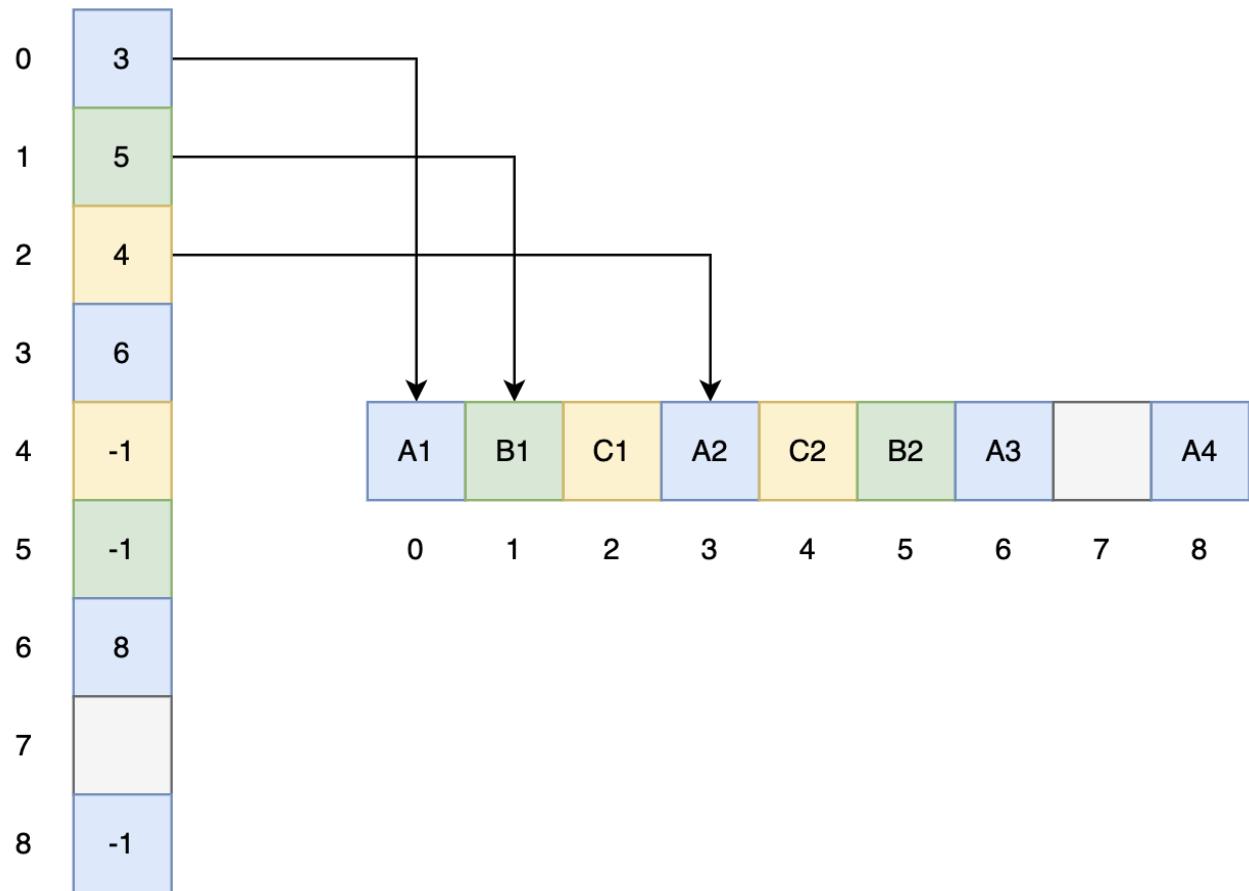


- velikost bloku = 1 KB
- velikost A = 3 KB
- velikost B = 3 KB

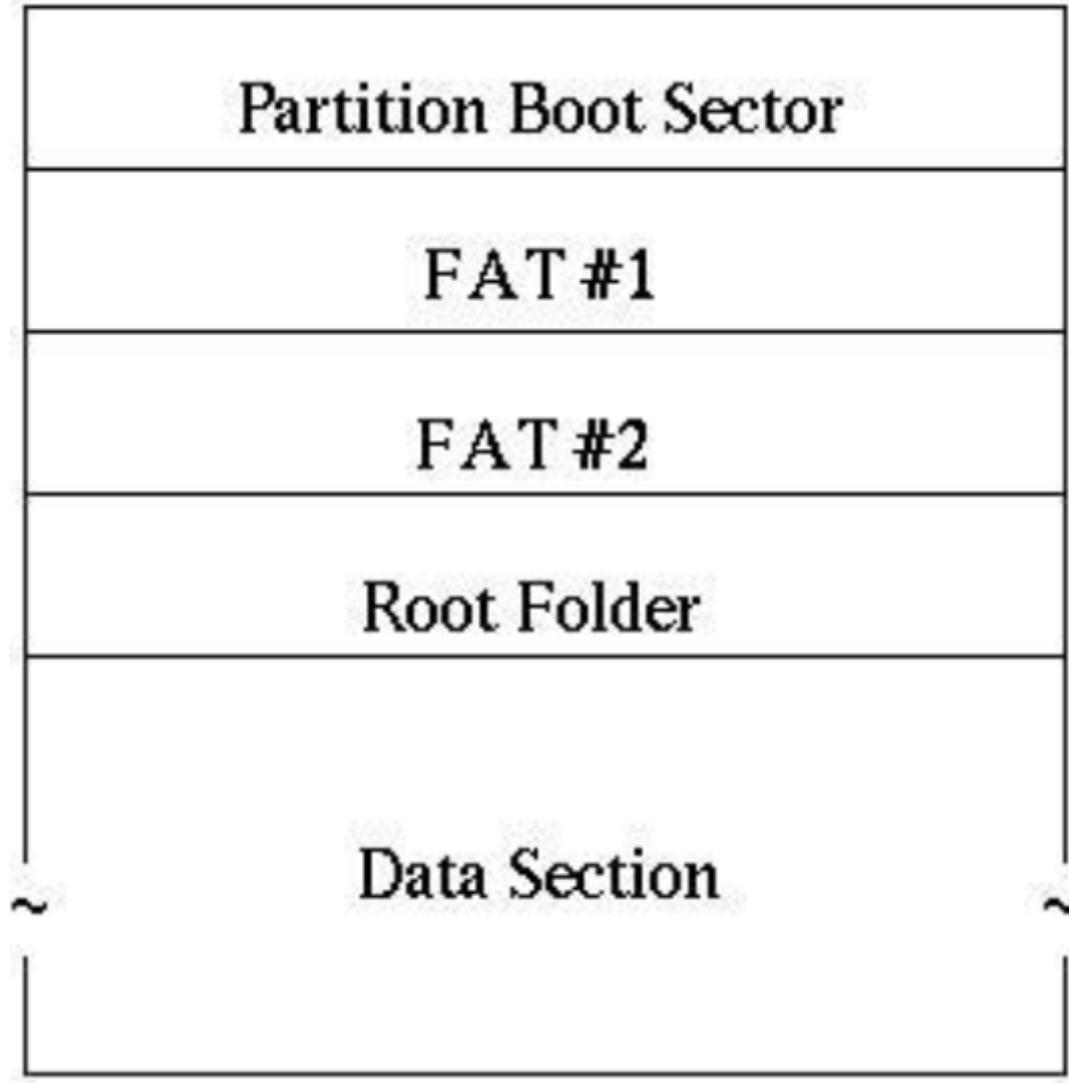
- velikost C = 2,5 KB

FAT

- **FAT** = file allocation table
- přesunutí odkazů do tabulky



- **defragmentace:**
 - **úplná** = obsazené bloky jsou za sebou
 - **částečná** = napřed obsazený prostor poté volné bloky
- diskový oddíl s FAT



- **problémy:**

- **fragmentace** = rozházené části souborů po disku
- **ztracené clustery** = clustery označeny jako používané ale nepatří
- **překřížené řetězy** = pro 2 a více souborů jsou vyhrazeny clustery se stejným číslem

NTFS

- nativní FS systému Windows

- **implementuje:**

- žurnálování
- access control list
- komprese
- šifrování
- diskové kvóty
- pevné a symbolické linky

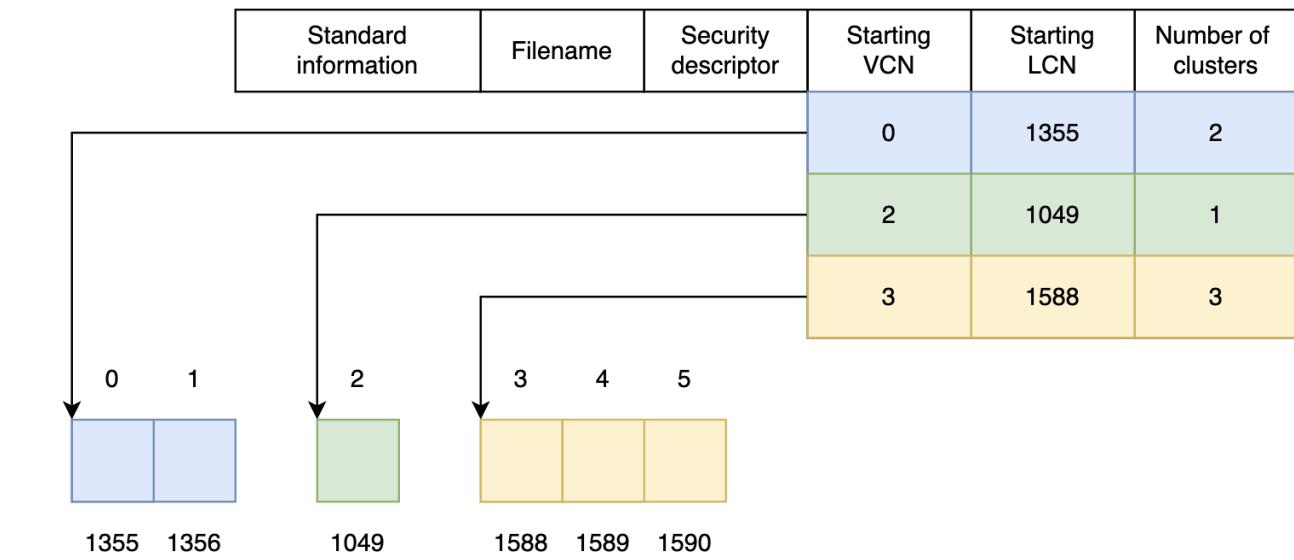
- alternativní datové proudy (poznamky.txt:tajny.txt)

Struktura

- 64b adresy clusterů = 16 EB
- logické číslování clusterů
- systém jako obří DB
- **systémové soubory:**
 - **\$LogFile** – žurnálování
 - **\$MFT (Master File Table)** – záznamy o všech souborech, adresářích, metadatech (je za boot sektorem)
 - **\$MFTMirr** – kopie částí záznamů \$MFT (uprostřed diskové oblasti)
 - **\$Badclus** – seznam vadných clusterů
 - **\$Bitmap** – sledování volného místa
- **defaultní velikost clusterů** = 4KB

Způsob uložení dat

- kódování délkou běhu
- **fragment** = index + počet bloků daného fragmentu

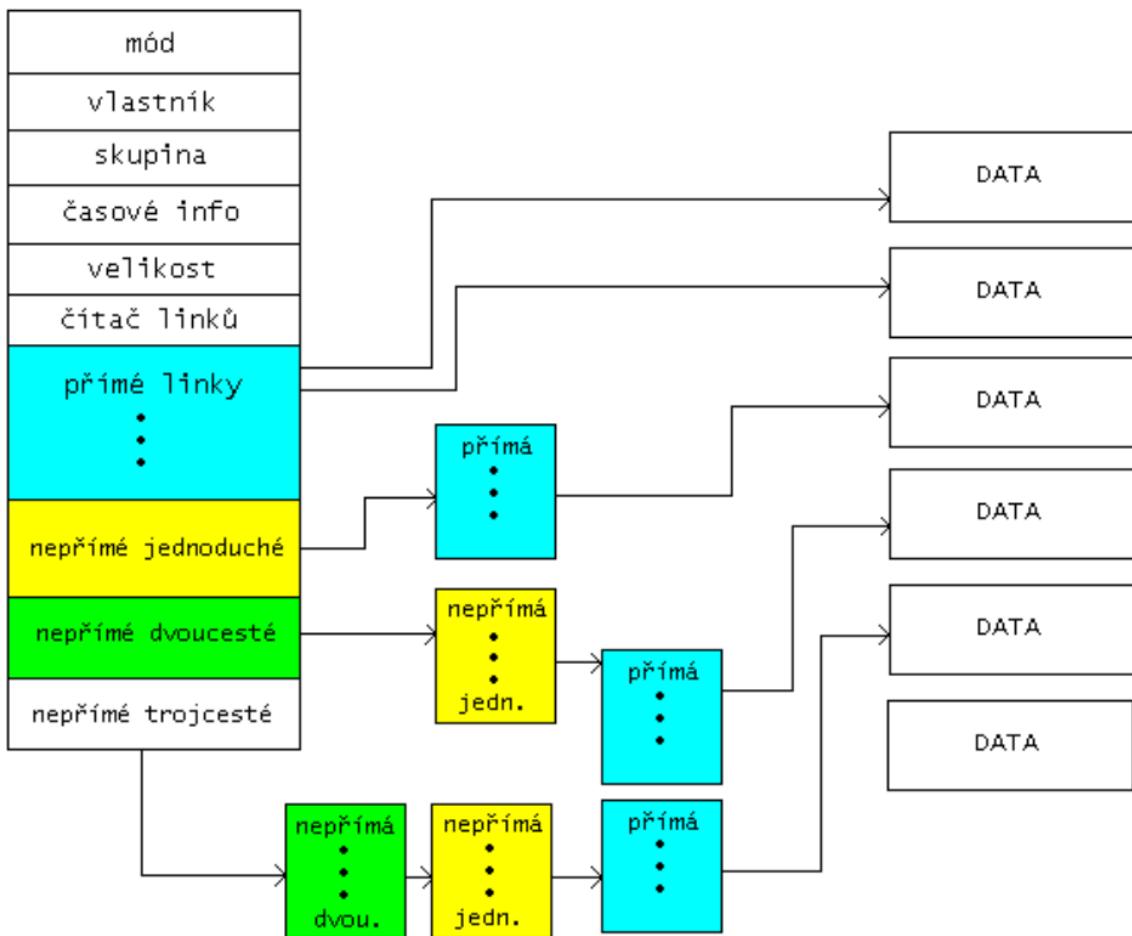


- - uloženy 3 soubory
 - ideálně 1 soubor = 1 fragment

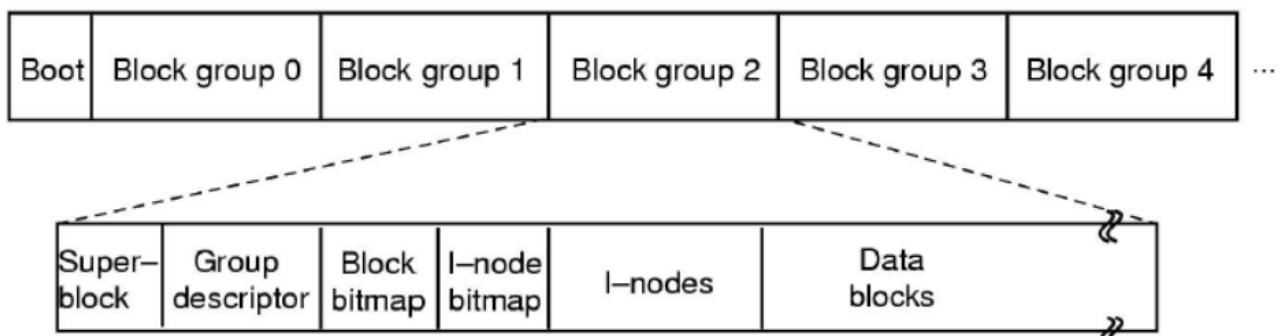
Systémy využívající i-uzlů

- každý soubor a adresář reprezentován i-uzlem
- **i-uzel** = datová struktura

- metadata
- umístění souboru na disku



Unixové systémy využívající i-uzly



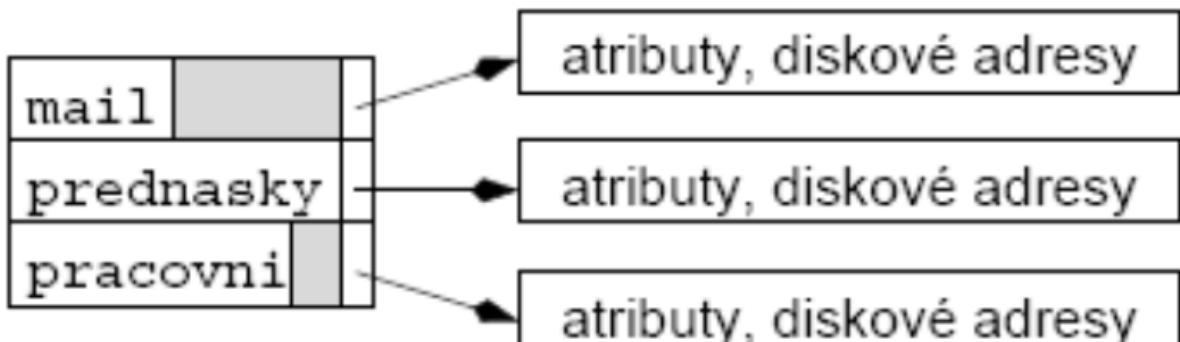
- **Block group** – obsahuje počet inodů a datových bloků, efektivnější zpráva velkých souborů
- **Groudn descriptor** – obsahuje informace o počtu volných a obsazených data bloků a inodů a ukazatele na bitmapy
- **Super block** = obsahuje velikost bloku, počet bloků, počet inodů a ukazatele na jejich pozice

Uspořádání adresáře

- adresář obsahuje jméno souboru, atributy, diskovou adresu souboru (používá FAT)

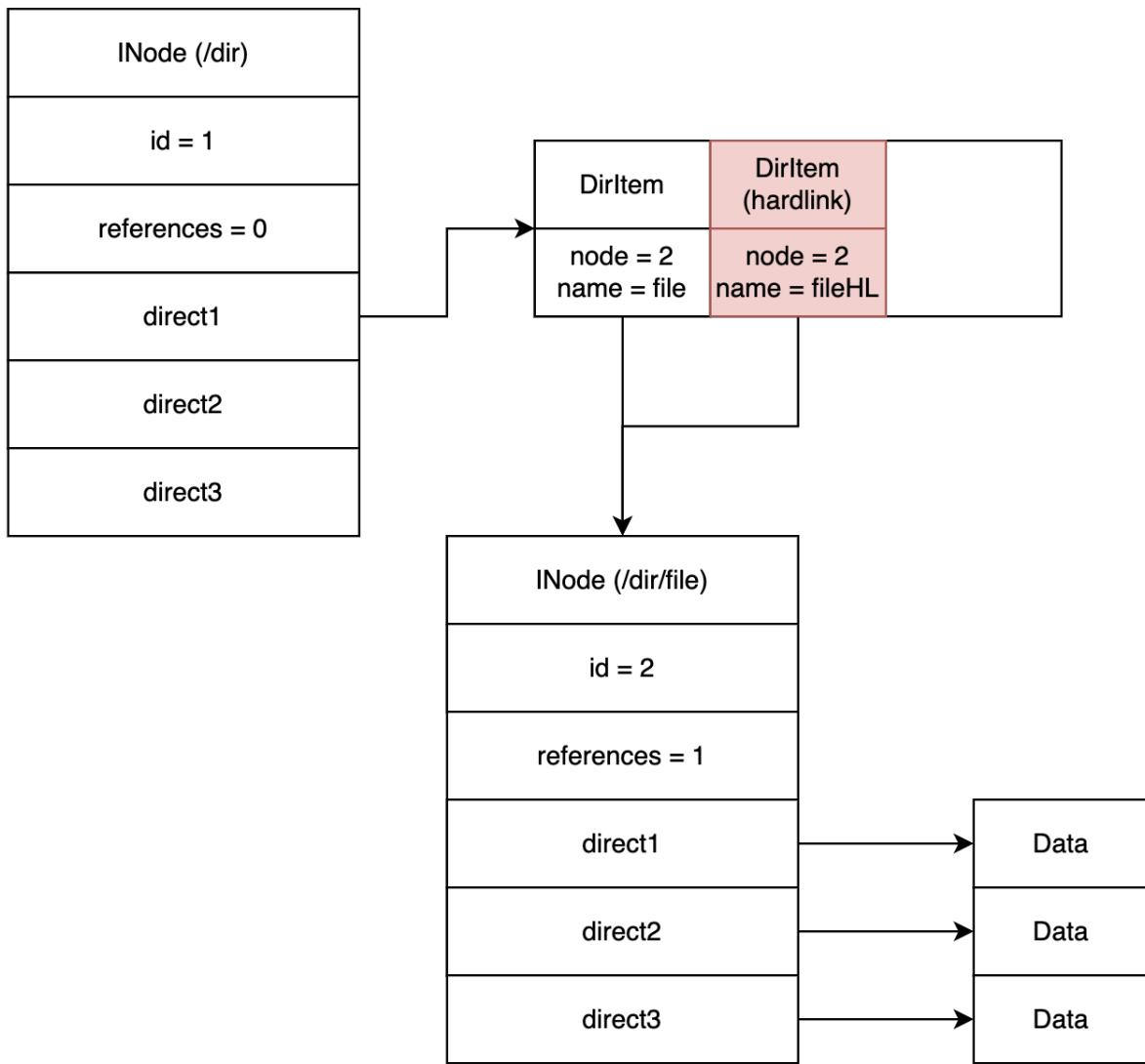
mail		atributy, diskové adresy
prednasky		atributy, diskové adresy
pracovni		atributy, diskové adresy

- adresář obsahuje pouze jméno + odkaz na jinou datovou strukturu (používá systém s i-uzly)

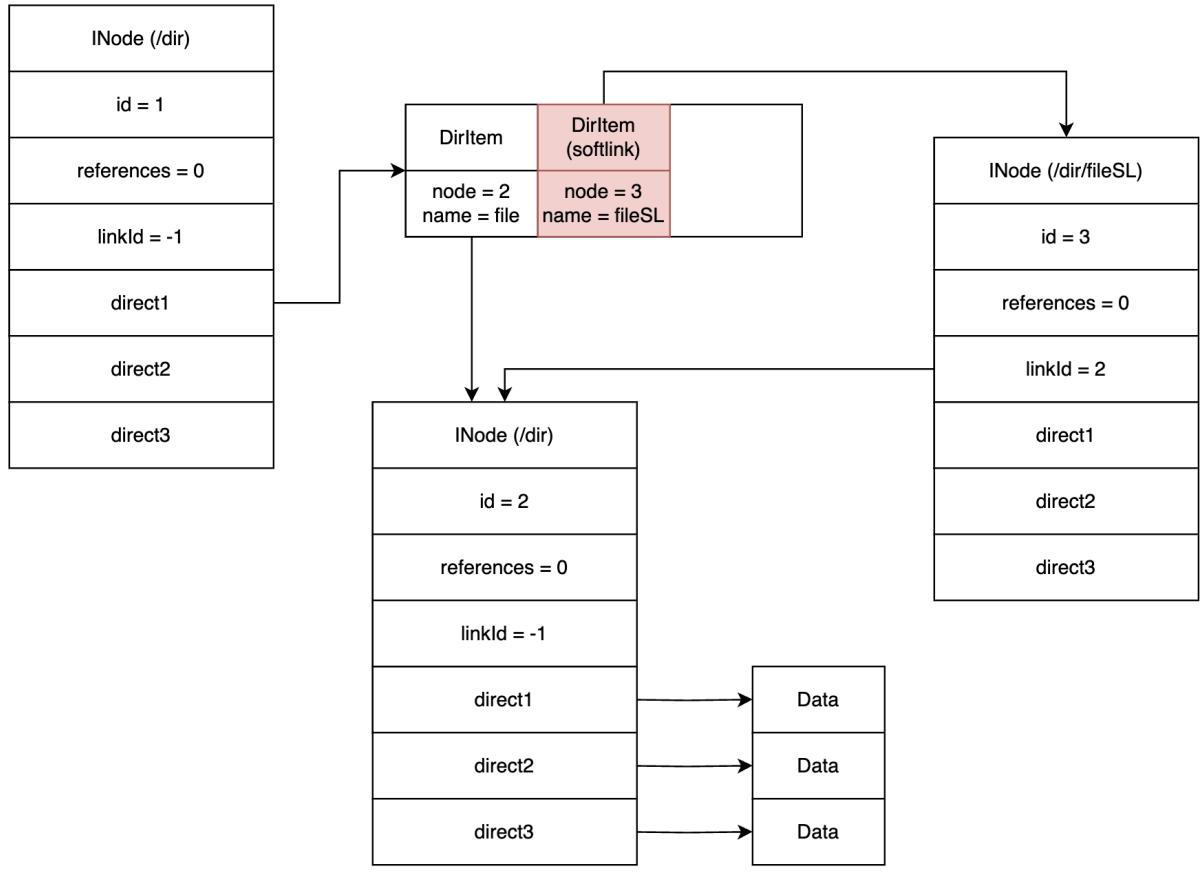


Hard link a soft link

- hard link:**



- **soft link:**



Kvóty

- **kvóty** = max. počet bloků obsazených soubory uživatele
- **druhy:**
 - **hard kvóta** = nelze překročit
 - **soft kvóta** = lze překročit => varování
 - **grace period** = doba, po kterou lze překročit soft kvótu

Testy konzistence FS

- **konzistentní FS:**

Číslo bloku	0	1	2	3	4	5	6	7	8
Výskyt v souborech	1	0	1	0	1	0	2	0	1
Volné bloky	0	1	0	0	1	0	0	1	0

Možné chyby:

chyba	popis	závažnost
(0,0)	missing block	malá
(0,2)	blok 2x v seznamu volných	malá
(1,1)	blok patří souboru a zároveň je mezi volnými	velká
(2,0)	blok patří do 2 souborů	největší

Journaling FS

- **princip Journaling FS:**

- před zápisem na disk se vytvoří na disku záznam popisující plánované operace
- poté se provedou operace a záznam se ruší

- **princip žurnálu:**

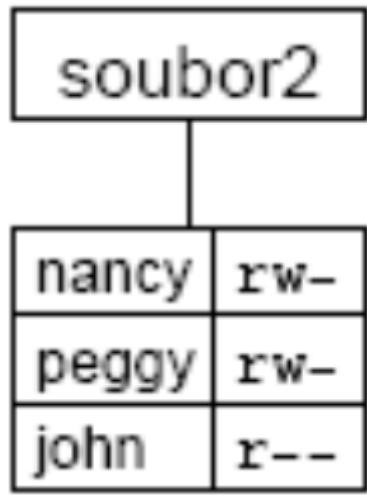
1. zapíši do žurnálu
2. žurnál kompletní?
 - ANO: zápis značky ZURNAL_KOMPLETNÍ
3. zapisování datových bloků
4. hotov => smazat žurnál

- **osetření výpadku:**

- FS se podívá do žurnálu:
 - prázdný => nic se nedělá
 - neprázdný ale není značka ZURNAL_KOMPLETNÍ => smazat žurnál
 - plný + je značka ZURNAL_KOMPLETNÍ => přepsání obsahu žurnálu do datových bloků

ACL (Access Control List)

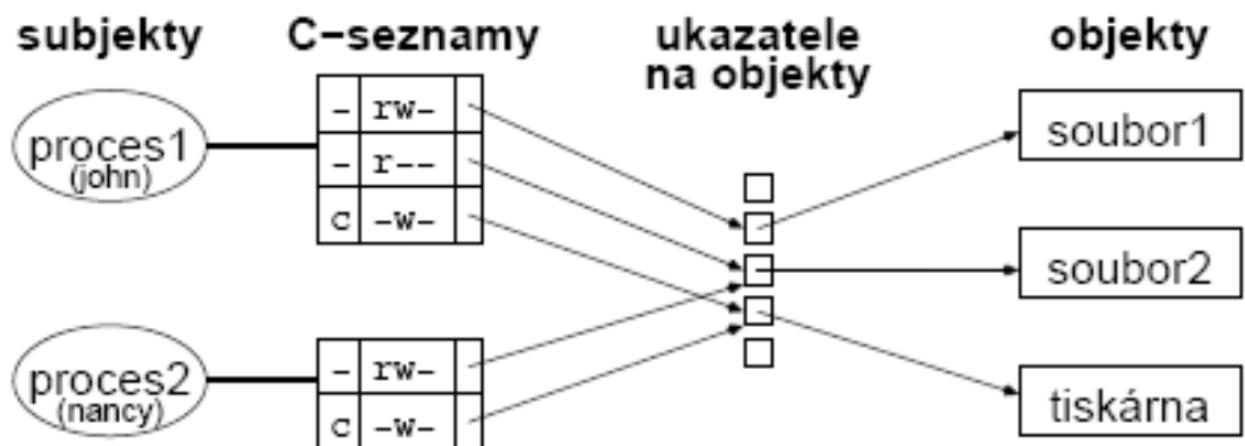
- **ACL** = množina přístupových práv k objektu



- má přednost před klasickými unixovými právy

Capability list

- zjednodušuje rušení přístupu k objektu



- pro odebrání přístupu stačí pouze smazat ukazatel

Správa I/O

- **důležité pojmy:**

- instrukce **IN**, **OUT** pro práci s periferním zařízením
 - privilegované

- používají adresy I/O portu – jiný adresní prostor než do RAM
- adresa 70 v IN instrukci je jiná než adresa 70 do RAM
- signál (drát v PC) M/IO určuje, zda je adresa na adresních vodičích do paměti nebo I/O portu

Vývoj rozhraní mezi CPU a zařízeními

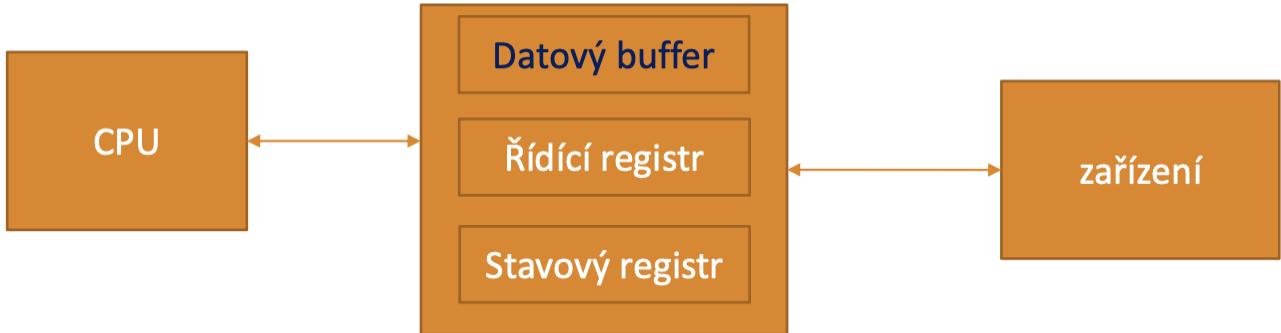
1. CPU řídí přímo periferii
2. CPU – řadič – periferie
3. řadič umí vyvolat přerušení
4. řadiš umí DMA
5. I/O modul
6. I/O modul s vlastní pamětí

1. CPU řídí přímo periferii

- CPU přímo vydává potřebné signály
- CPU dekóduje signály poskytované zařízením
- nejjednodušší HW
- nejméně efektivní využití CPU

2. CPU – řadič – periférie

- **řadič (device controller)**
 - převádí příkazy CPU na elektrické impulzy pro zařízení
 - poskytuje CPU info o stavu zařízení
 - komunikace s CPU pomocí registrů řadiče na známých I/O adresách
 - HW buffer pro alespoň 1 záznam
 - **povel** = příkaz, který dává CPU řadiči
 - **stav** = informace o stavu
 - **data** = buffer předává data



řadič

- příklad operace zápisu:
 - CPU zapíše data do bufferu
 - CPU informuje řadič o požadované operaci
 - řadič po dokončení výstupu zařízení nastaví příznak, který může CPU otestovat
 - if přenos == OK může vložit další data
 - CPU musí dělat všechno
 - významnou část času stráví CPU čekáním na dokončení I/O operace

3. Řadič umí vyvolat přerušení

- CPU nemusí testova příznak dokončení
- při dokončení I/O vyvolá řadič přerušení
- CPU začne obsluhovat přerušení
- postačuje pro pomalá zařízení

4. Řadič může přistupovat k paměti pomocí DMA

- DMA přenosy mezi pěmtí a I/O zařízením
- CPU inicializuje přenos, ale sám ho nevykoná
- **řadič DMA** = speciální obvod, zajišťuje blokové přenosy mezi I/O zařízením a pamětí
- **princip:**
 1. CPU zadá požadavek řadiči DMA
 - odkud: adresa I/O zařízení
 - kam: adresa v RAM
 - kolik: počet bytů
 2. DMA obvod provede přesun dat bez zásahu CPU
 3. CPU se zatím může věnovat dalším věcem

4. po ukončení přenosu DMA obvod vyvolá přerušení

5. I/O modul umí interpretova speciální I/O programy

- I/O procesor
- interpretuje program v RAM
- CPU pustí I/O procesor
 - I/O procesor provádí své instrukce samostatně
 - v podstatě samostatný PC

6. I/O modul s vlastní pamětí

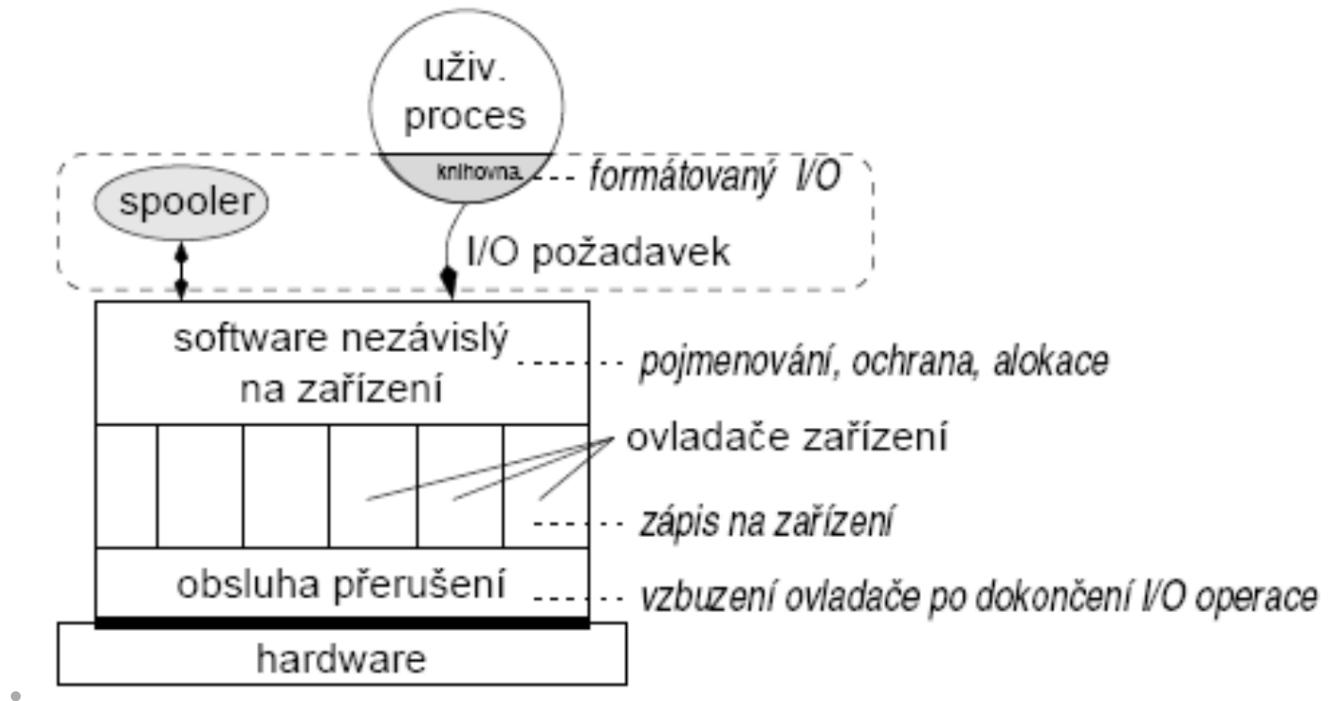
- I/O modul provádí programy
- má vlastní paměť => je vlastně samostaným PC

Komunikace CPU s řadičem

- **odlišené adresní prostory (I/O prostor)**
 - CPU zapisuje do registrů řadiče pomocí speciálních I/O instrukcí
 - **vstup:** IN R, port
 - **výstup:** OUT R, port
- **1 adresní prostor (RAM)**
- **hybridní schéma (I/O, RAM)**

Principy I/O SW

-
1. obsluha přerušení
 2. ovladač zařízení
 3. SW vrstva OS nezávislá na zařízení
 4. uživatelský I/O SW



1. Obsluha přerušení

- řadič vyvolá přerušení ve chvíli dokončení I/O požadavku
- snaha aby se přerušením nemusely zabývat vyšší vrstvy
- ovladač zadá I/O požadavek, usne – P(sem)
- po příchodu přerušení ho obsluha přerušení vzbudí – V(sem)
- časově kritická obsluha přerušení

2. Ovladače zařízení

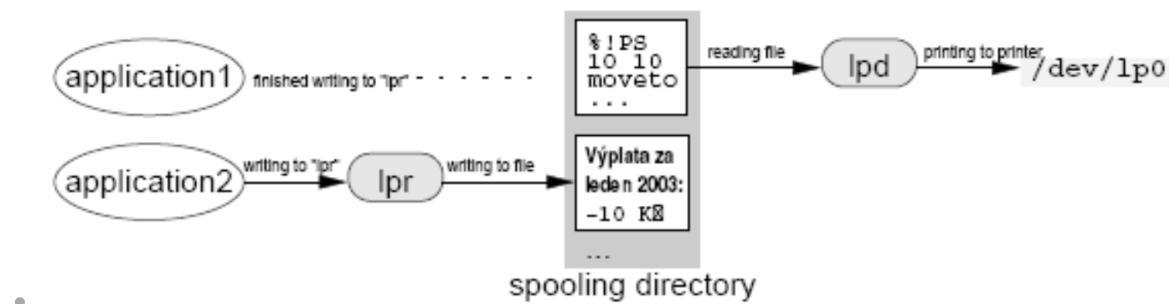
- obsahují veškerý kód závislý na konkrétním I/O zařízení
- ovladač zná jediný HW podrobnosti
- může ovládat všechna zařízení daného druhu nebo třídu příbuzných zařízení
- **funkce:**
 1. ovladači předán příkaz od vyšší vrstvy
 2. nový požadavek zařazen do fronty
 3. ovladač zadá příkazy řadiči
 4. zablokuje se do vykonání požadavku
 5. vzbuzení obsluhou přerušení (dokončení operace) – zkontroluje zda nenastala chyba
 6. pokud OK, předá výsledek (status + data) vyšší vrstvě
 7. Zpracuje další požadavky ve frontě

3. SW vrstva OS nezávislá na zařízení

- I/O funkce společné pro všechna zařízení daného druhu
- definuje rozhraní s ovladačí
- poskytuje jednotné rozhraní uživatelskému SW
- **funkce:**
 - pojmenování zařízení
 - ochrana zařízení
 - alokace a uvolnění vyhrazených zařízení
 - vyrovnávací paměť
 - hlášení chyb
 - jednotná velikost bloku

4. I/O sw v uživatelském režimu

- programátor používá v programech I/O funkce nebo příkazy jazyka
- **spooling:**
 - implementován pomocí procesů běžících v uživatelském režimu
 - způsob obsluhy vyhrazených I/O zařízení
 - např. proces by alokoval zařízení a pak hodinu nic nedělal
- **příklad spoolingu:**
 - přístup k fyzické tiskárně – pouze 1 speciální proces (daemon lpd)
 - proces vygeneruje celý soubor, lpd ho vytiskne
 - proces chce tisknout, spustí lpr a naváže s ním komunikaci
 - proces předává tisknutá data programu lpr
 - lpr zapíše data do souboru v určeném adresáři
 - **spooling directory** – přístup jen lpr a lpd
 - dokončení zápisu – lpr oznámí lpd, že soubor je připraven k vytisknutí
 - lpd soubor vytiskne a zruší



Disk

- **rotační disky**
 - doba vystavení (posun hlaviček) + rotační zpoždění
 - stopa, sektor
- **SSD disk**
 - dražší, menší kapacita

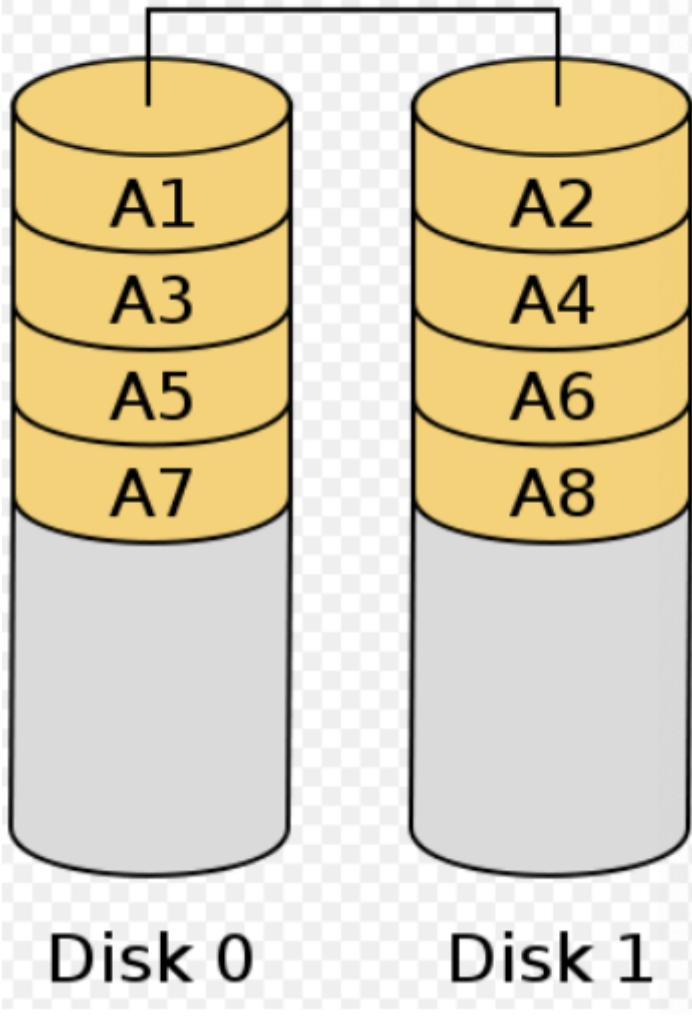
RAID

- **RAID** = Redundant Array of Independent Disks
- data distribuovaná na více disků
- sada fyzických disků, OS je následně vidí jako jeden disk

RAID 0

- není redundanční, neposkytuje ochranu dat
- ztráta 1 disku – ztráta celého pole nebo části dat
- **režimy:**
 - **zřetězení:**
 - data postupně ukládána na několik disků
 - zaplní se první disk, pak druhý, atd.
 - snadné zvýšení kapacity
 - **prokládání:**
 - data ukládána na disky cyklicky po blocích
 - při poruše jednoho disku přijdeme o data
 - větší rychlosť čtení/zápisu

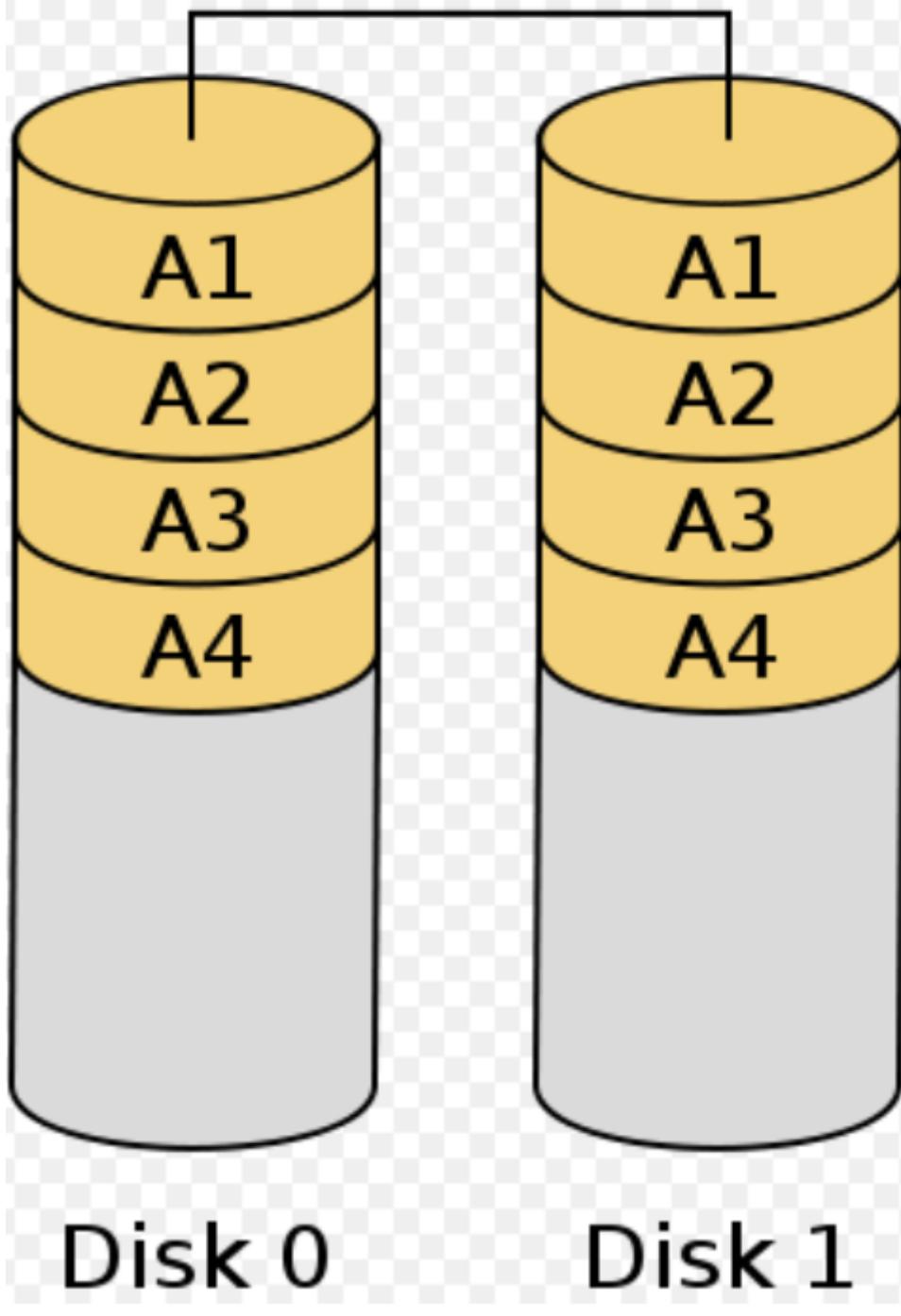
RAID 0



RAID 1

- mirroring – zrcadlení
- na 2 disky stejných kapacit ukládáme totožné data
- výpadek 1 disku nevadí
- jednoduchá implementace
- nevýhoda – polovina kapacity
- zápis – pomalejší
- čtení – rychlejší

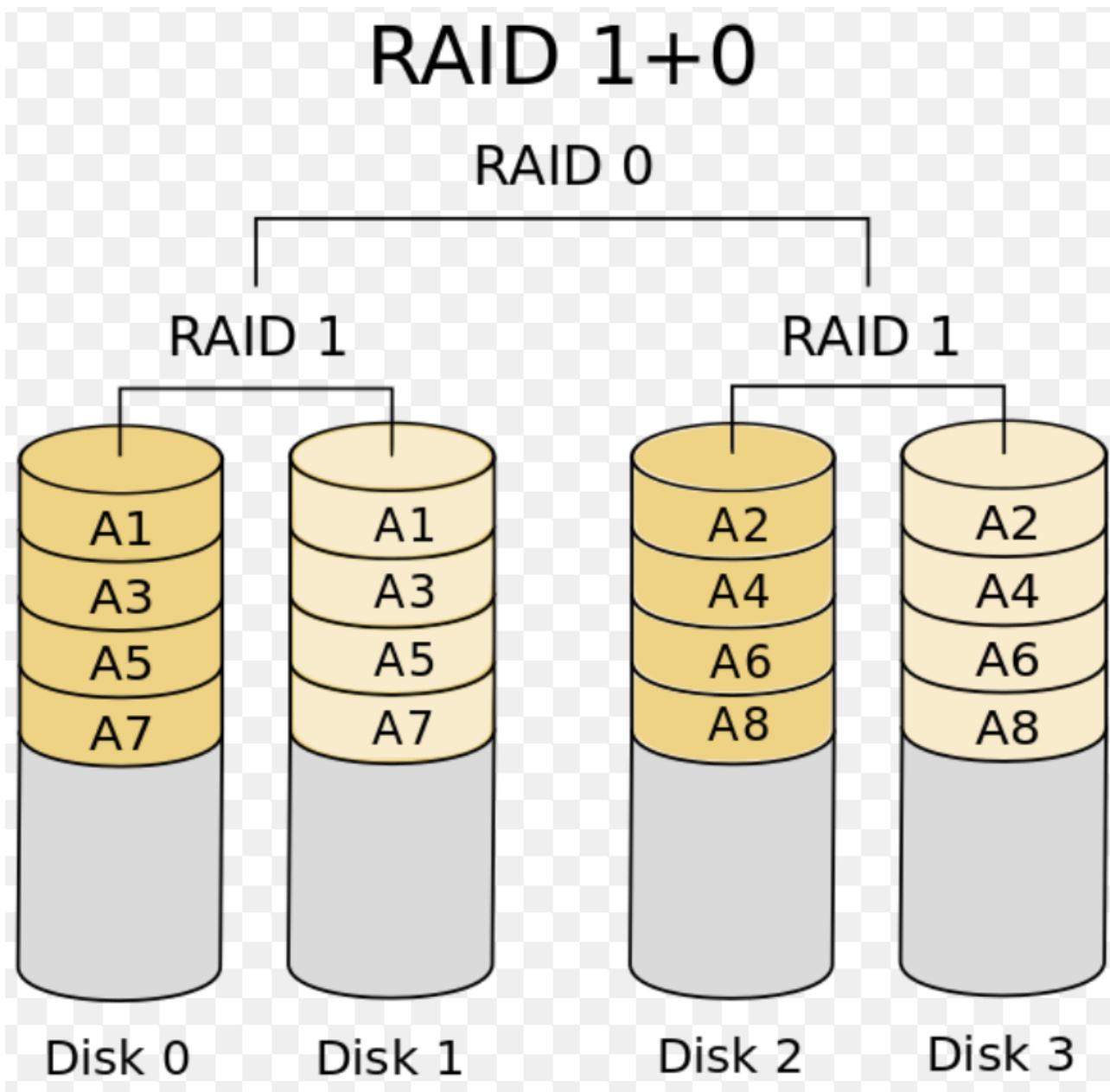
RAID 1



RAID 10

- kombinace RAID 0 (stripe) a RAID 1 (zrcadlo)
- min. 4 disky
- režie 100% diskové kapacity navíc
- rychlejší než RAID 5 při zápisu

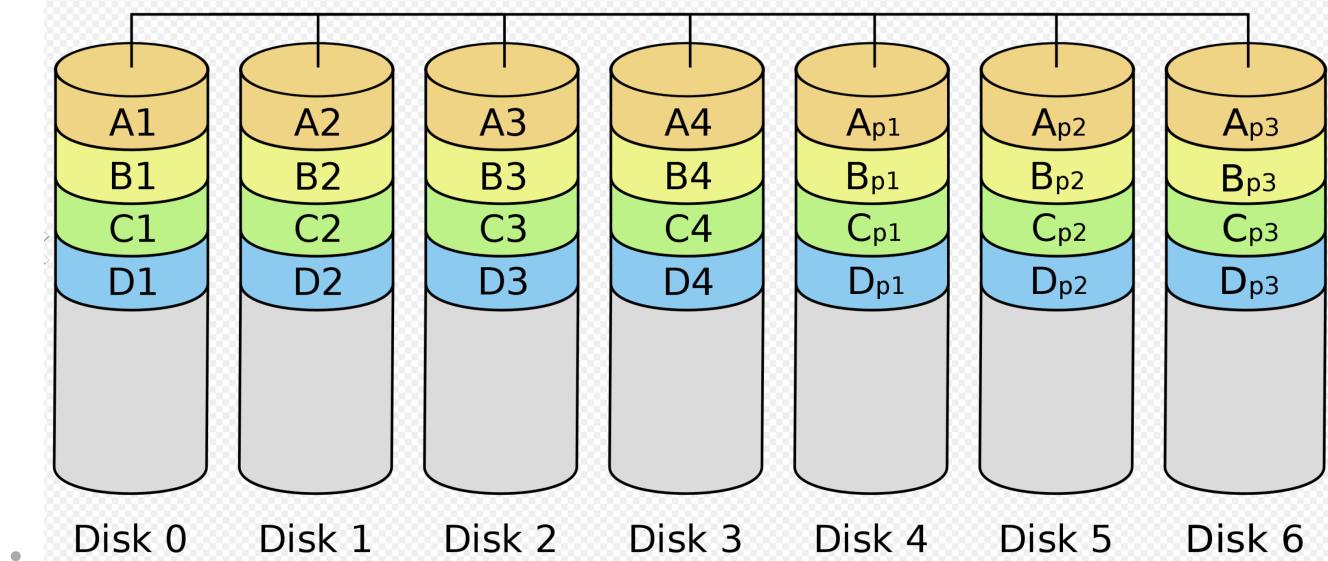
- odolnost proti ztrátě až 50% disků



RAID 2

- data po bitech stripována mezi jednotlivé disky
- zabezpečení hammingovým kódem

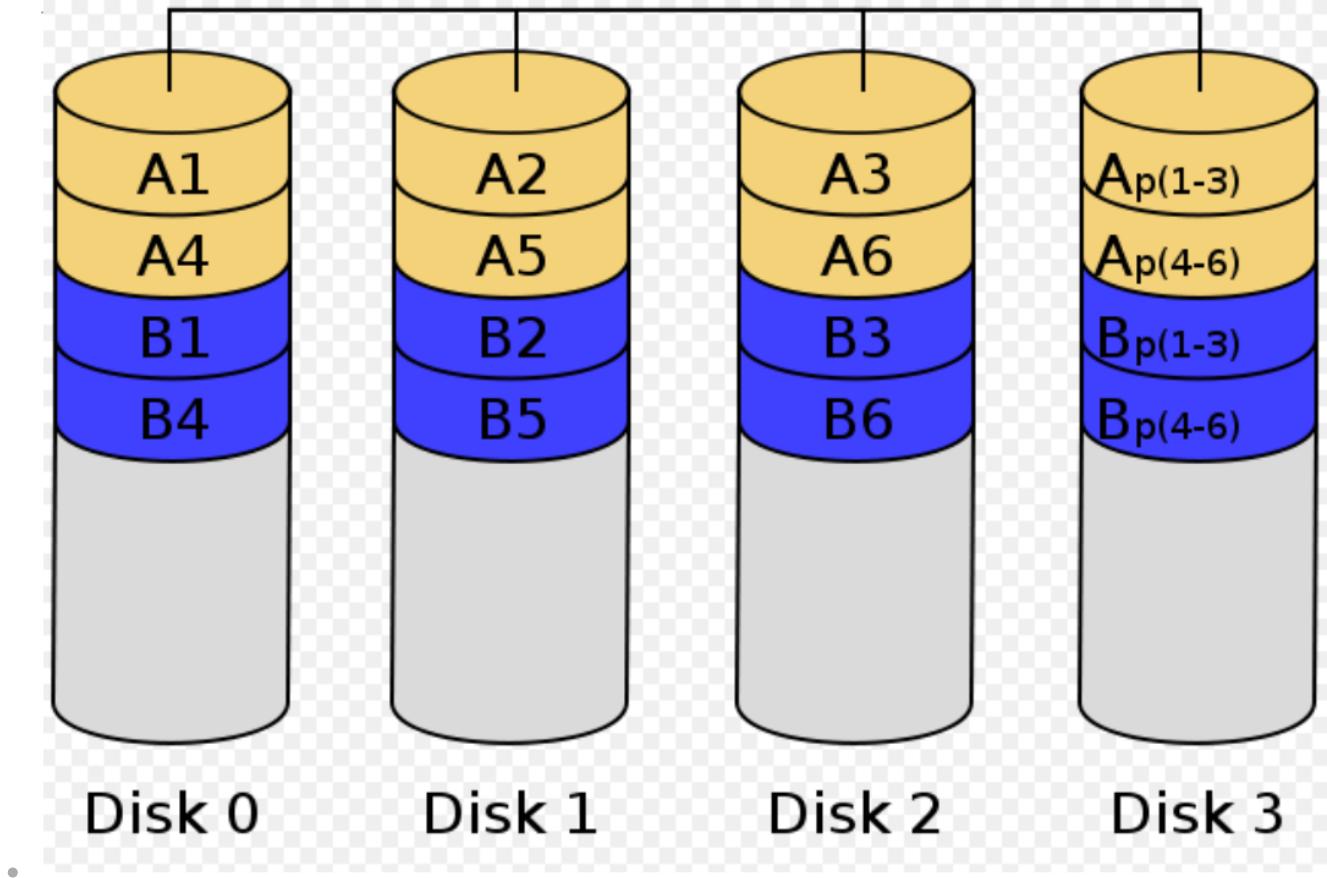
RAID 2



RAID 3

- N+1 disků, bitové prokládání
- na N data, poslední disk XOR parita
- jen 1 disk navíc
- paritní disk je více vytížen

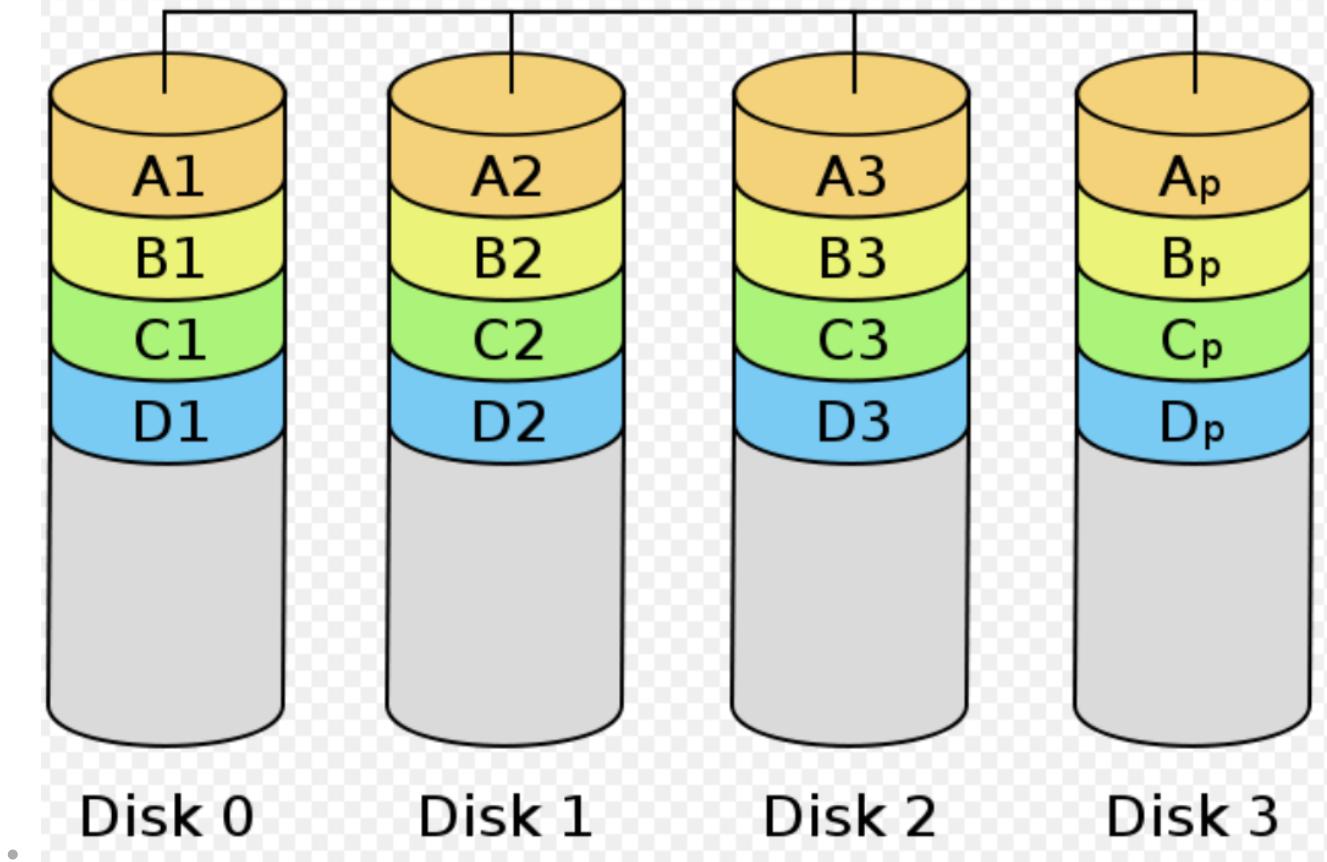
RAID 3



RAID 4

- disky stripovány po blocích, ne po bitech
- parita opět po blocích

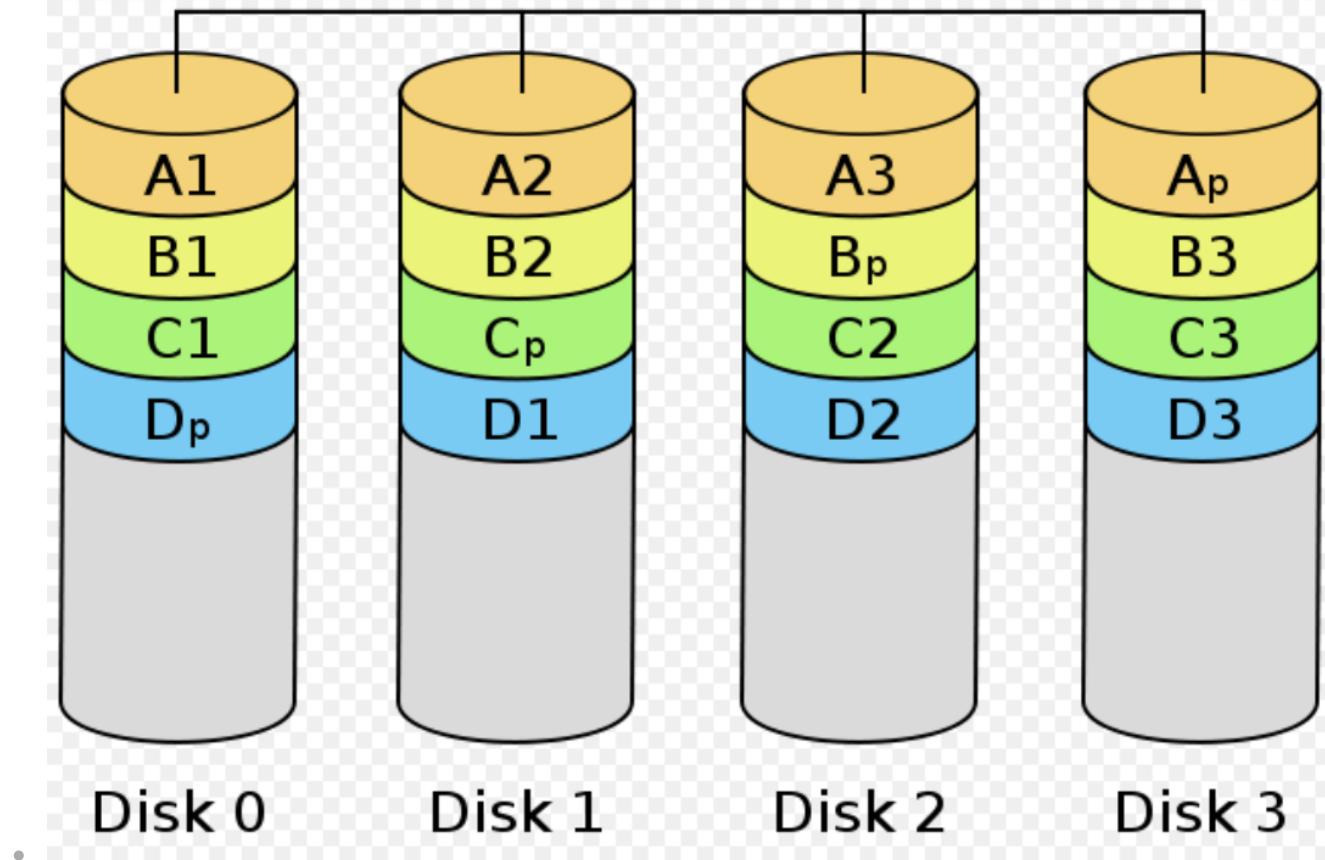
RAID 4



RAID 5

- redundantní pole s distribuovanou paritou
- minimálně 3 disky
- režie: odpovídá kapacitě 1 disku z pole n disků
 - 5 disků 100GB: 400GB pro data
- výpadek 1 disku nevadí
- čtení – výkon OK
- zápis – pomalejší

RAID 5



- **princip:**
 - máme 3 disky a blok má 3 byty
 - červeně je označen paritní bit
- **výpadek disku 2:**

Disk 1	Disk 2	Disk 3	Co ukládáme například:
1	0	1	1 0
0	1	1	0 1
0	0	0	0 0
1	1	0	1 1

Disk 1	Disk 2	Disk 3	Co ukládáme například:
1	?	1	1 0
0	?	1	0 1
0	?	0	0 0
1	?	0	1 1

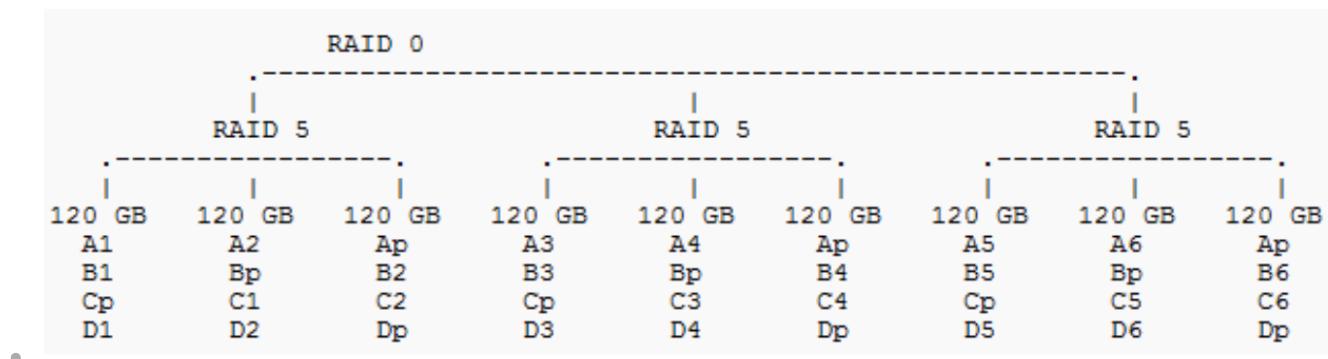
V každém řádku dopočteme, aby mezi sloupcí disk1, 2, 3 byl sudý počet jedniček

Disk 1	Disk 2	Disk 3	Co ukládáme například:
1	0	1	1 0
0	1	1	0 1
0	0	0	0 0
1	1	0	1 1

- **degradovaný režim:**

- jeden disk vypadne – uživatel nepozná, pole běží v degradovaném režimu (maskuje poruchu)

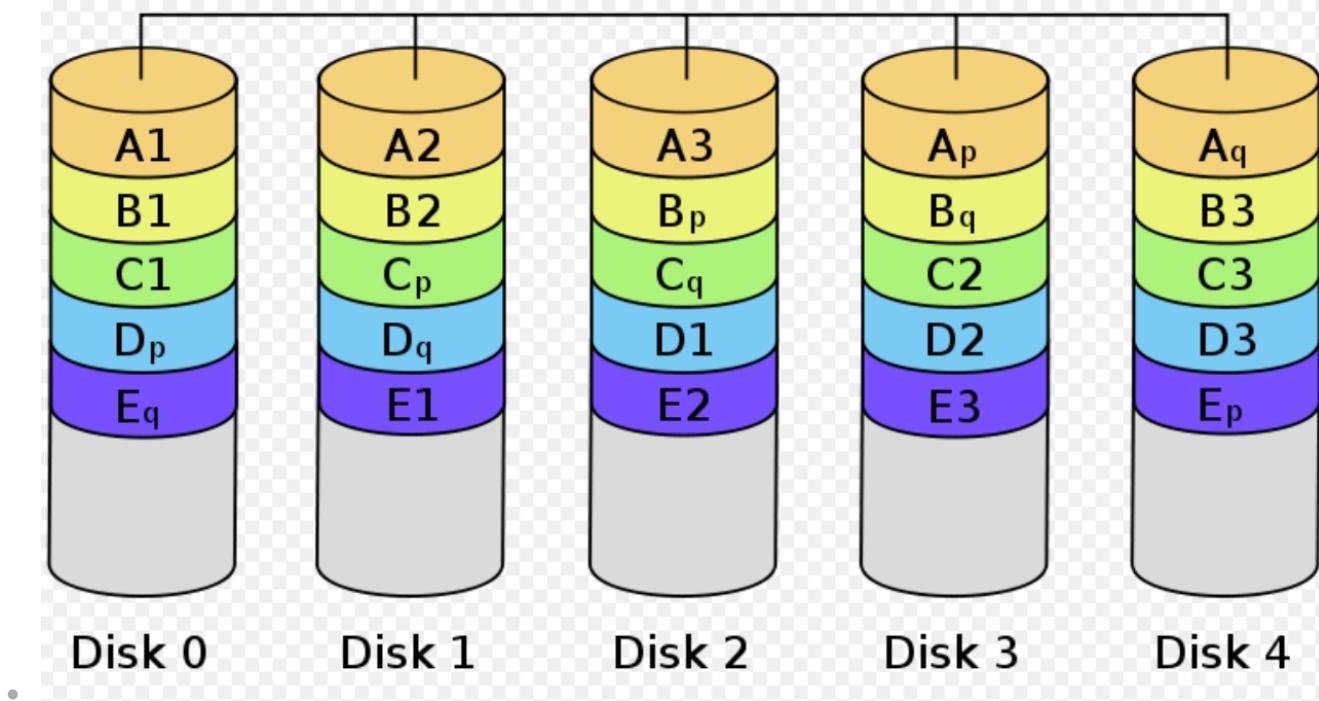
RAID 50



RAID 6

- RAID 5 + navíc další pravítní disk
- odolné proti výpadku 2 disků
- rychlosť čtení jako RAID 5
- zápis pomalejší

RAID 6



Problém rekonstrukce pole

- rekonstrukce trvá dlouho
- po dobu rekonstrukce není pole chráněno proti výpadku dalšího disku

HOT SPARE DISK

- záložní disk okamžitě připravený k nahrazení vadného disku
- při výpadku disku v poli automaticky aktivován hot spare disk a dopočítána data

HOT PLUG

- snadná výměna disku za běhu systému
- není třeba vypnout server pro výměnu disku
- "šuplík z přední strany serveru"

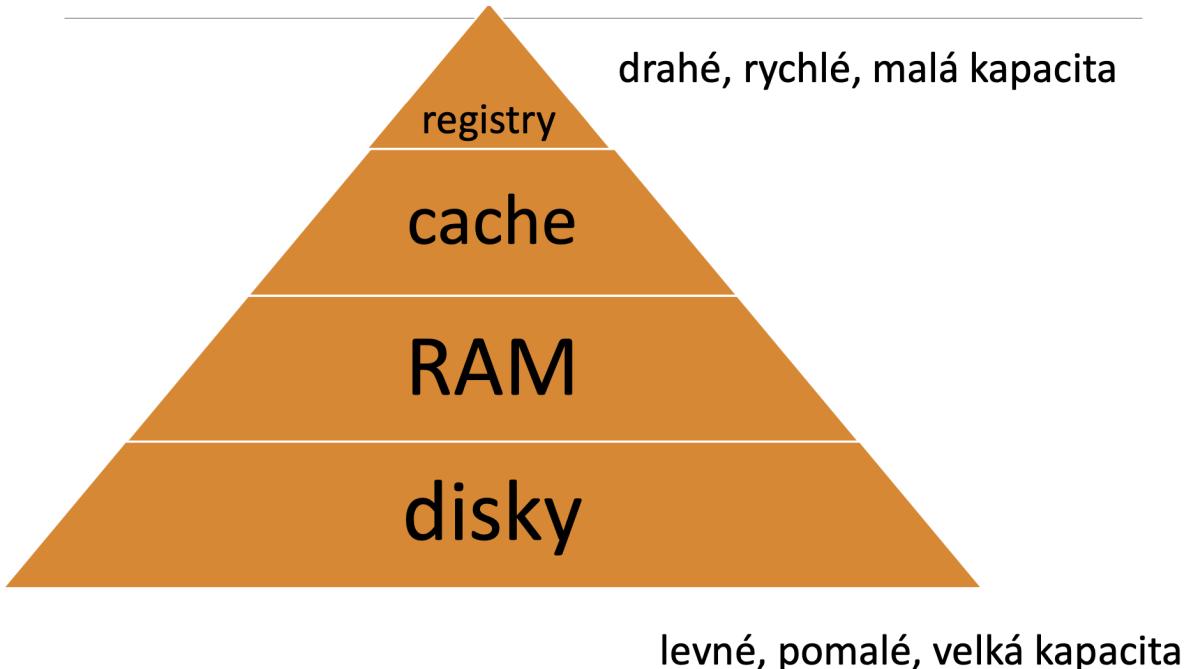
Správa paměti

- ideál – paměť nekonečně velká

- reálný PC – hierarchie pamětí

- registry CPU
- cache paměť
- RAM
- pevné disky

- **paměťová pyramida:**



Správce paměti

- **správce paměti** = část OS spravující paměť
- udržuje info o alokaci
- alokuje paměť procesům (**malloc**)
- zařazuje paměť do volné paměti po uvolnění procesem (**free**)
- paměť se alokuje z heapu
- **příklad alokace:**
 1. proces chce 500B zavolá `malloc`
 2. alokátor zkонтroluje, že nemá volnou paměť, požádá OS o přidělení stránky paměti (4KB) – `sbrk`
 3. proces dostane 500B
 4. proces chce dalších 200B zavolá `malloc`
 5. alokátor už paměť v sobě má rovnou jí přidělí procesu
 6. když proces paměť nepotřebuje volá `free`
- ukazatel = `malloc(size)` – obsahuje virtuální adresu, není to adresa do RAM

Mechanismy správy paměti

- **kategorie:**

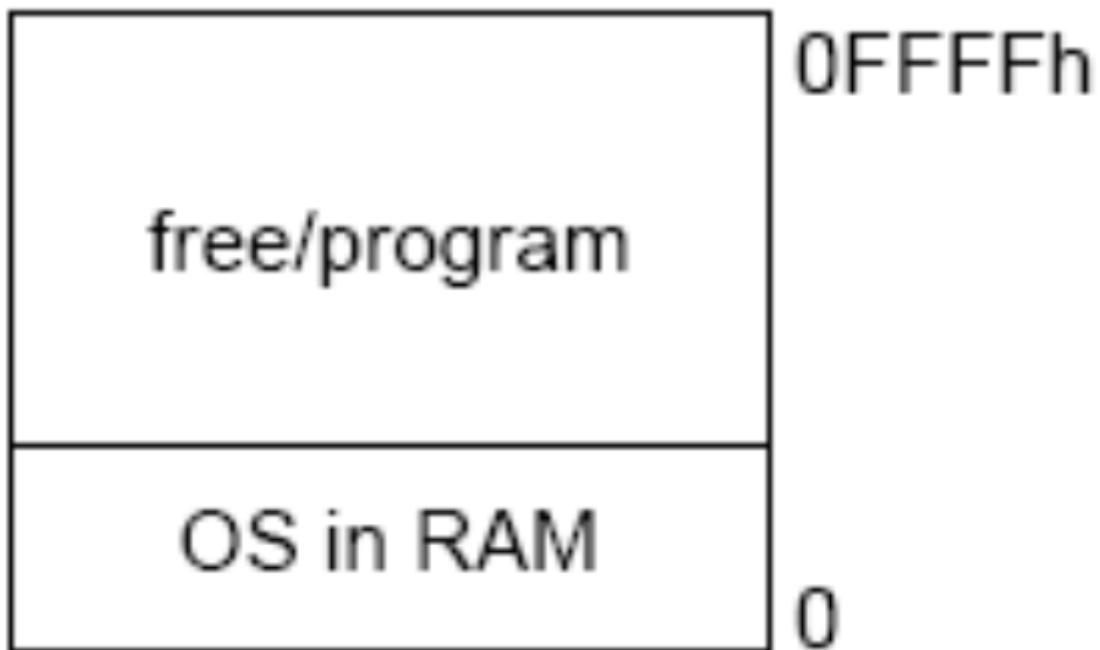
- základní mechanismy – program je v paměti celou dobu běhu
- mechanismy s odkládáním – programy přesouvá mezi hlavní pamětí a diskem

Základní mechanismy pro správu paměti

1. jednoprogramové systémy
2. Multiprogramování s pevným přidělením paměti
3. Multiprogramování s proměnnou velikostí oblasti

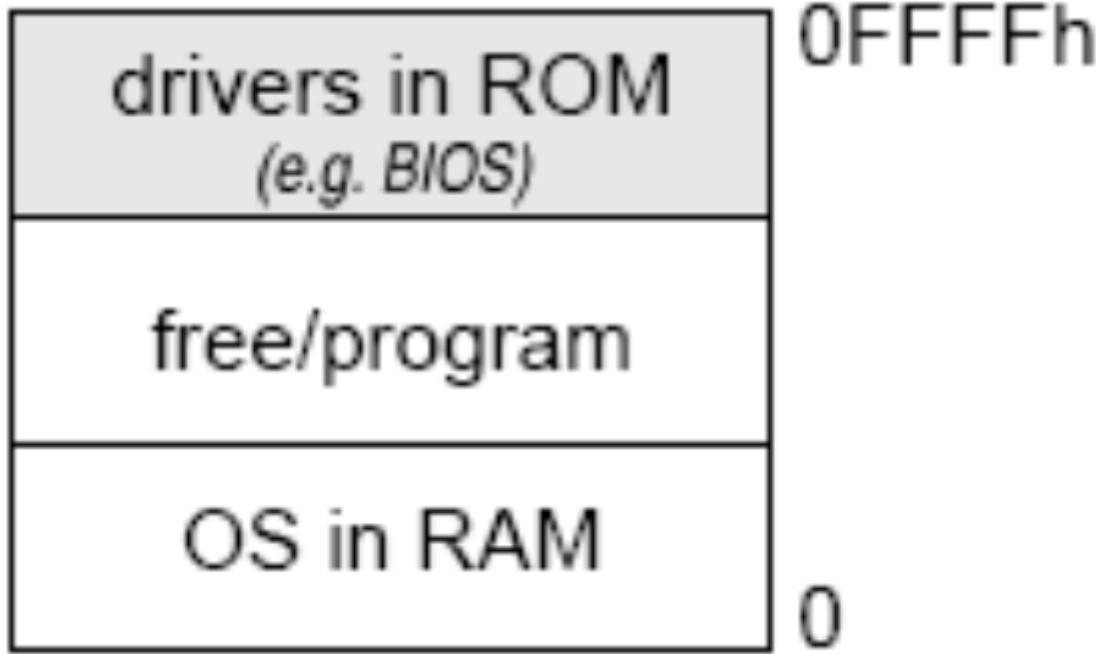
Jednoprogramové systémy

- pouze jeden program v jendom čase
- program používá veškerou paměť, kterou nepotřebuje OS
- **dělení:**
 - OS ve spodní části adresního prostoru RAM (miniPC)



- OS v horní části adresního prostoru ROM (zapozdřené systémy)

- OS v RAM, výchozí obslužné rutiny v ROM (na PC – MS DOS v RAM, BIOS v ROM)



Multiprogramování s pevným přidělením paměti

- paralelní nebo pseudoparalelní běh více procesů
- nejjednodušší schéma – rozdělit paměť na n oblastí
- multiprogramování zvyšuje využití CPU
 - proces – část času p tráví čekáním na dokončení I/O
 - N procesů – ppst, že všechny čekají na I/O je: p^n
 - **využití CPU je:** $u = 1 - p^n$

- n – stupeň multiprogramování
- **příklad:**
pokud proces tráví 80% času čekáním, $p = 0.8$

$n = 1 \dots u = 0.2$ (20% času CPU využito)

$n = 2 \dots u = 0.36$ (36%)

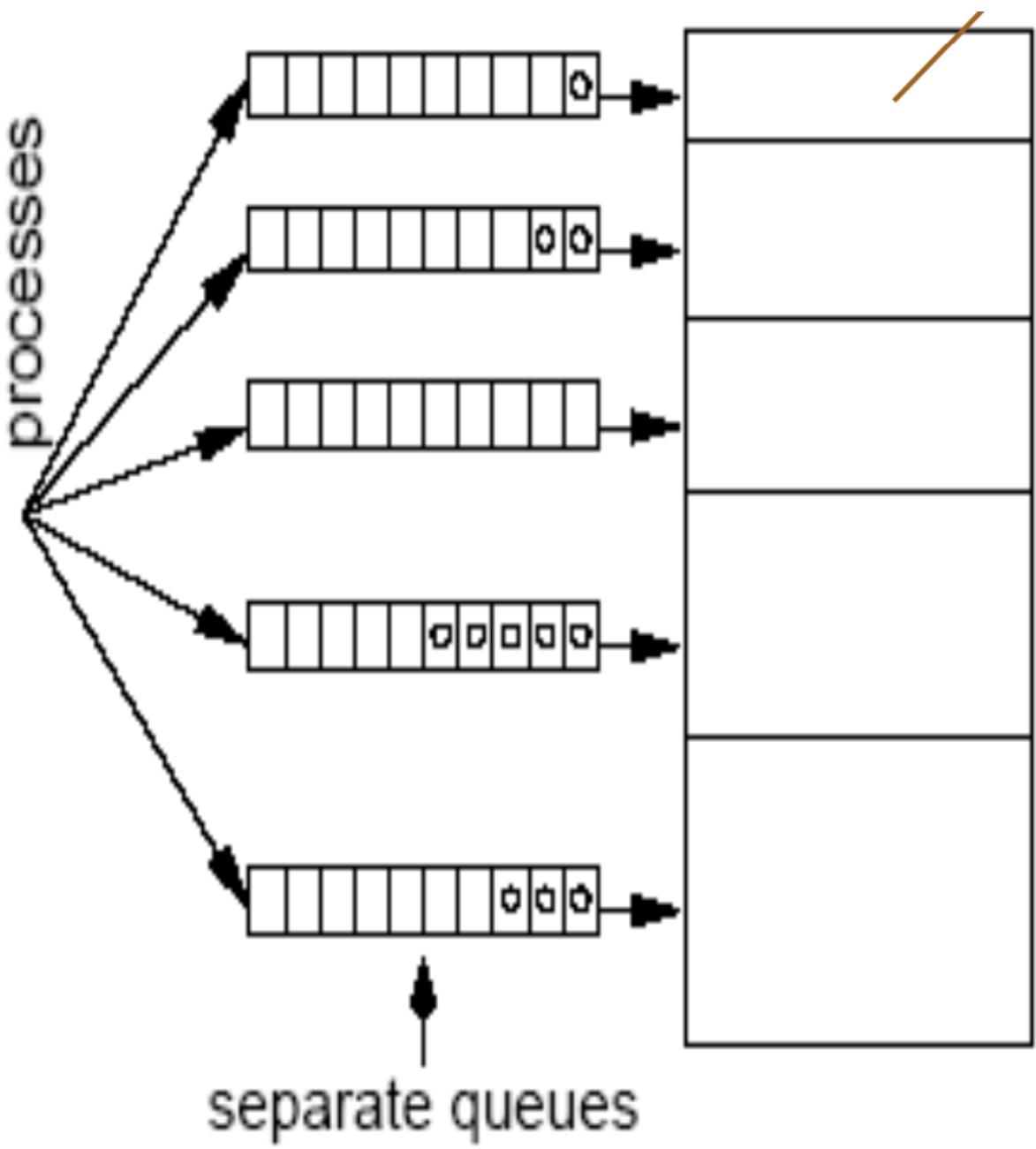
$n = 3 \dots u = 0.488$ (49%)

$n = 4 \dots u = 0.5904$ (59%)

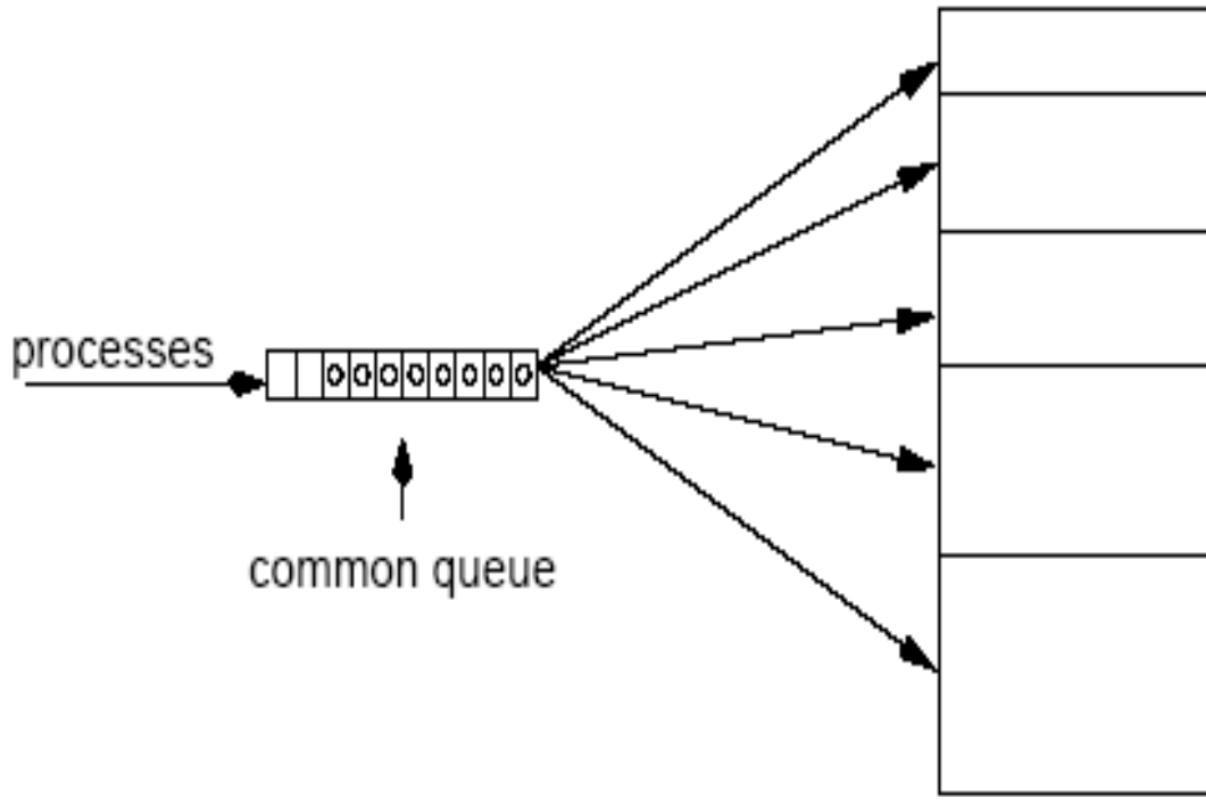
•

Pevné rozdělení sekcí

1. **více front** – každá úloha do nejmenší oblasti kam se vejde



- 2. **jedna fronta** – po uvolnění oblasti z fronty vybrat největší úlohu, která se vejde (**není FIFO**)



Multiprogramování s proměnnou velikostí oblasti

- úloze přidělena paměť dle požadavku
- v čase se mění:
 - počet oblastí
 - velikost oblastí
 - umístění oblastí
- zlepšuje využití paměti
- komplikovanší alokace/dealokace

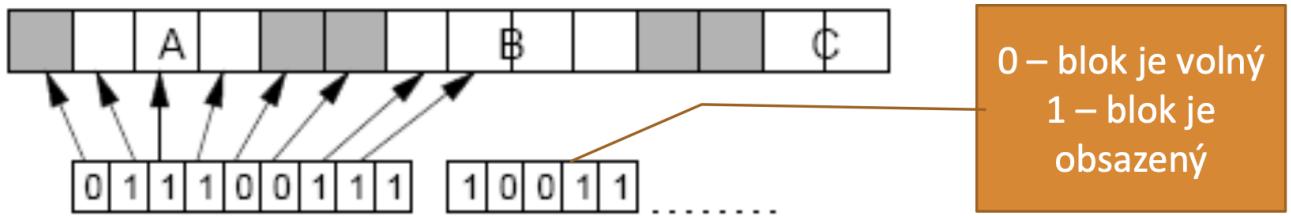
Volná X alokovaná paměť

- pro správu paměti se používá:
 1. bitové mapy
 2. seznamy
 3. buddy systems

Správa pomocí bitmap

- paměť rozdělena na alokační jednotky stejné délky

- s každou jednotkou 1bit (volno x obsazeno)



•

- **výhody:**

- konstantní velikost bitové mapy

- **nevýhody:**

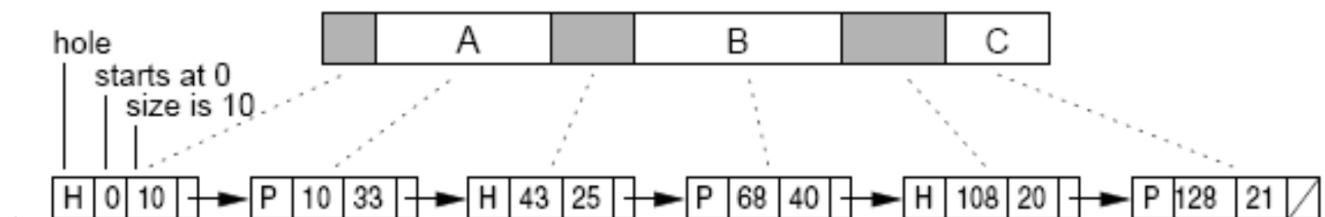
- najít požadovaný úsek N volných jednotek
- náročné nepoužívá se pro RAM

Správá pomocí seznamů

- seznam alokovaných a volných oblastí (procesů, dír)

- **položka seznamu:**

- info o typu – proces nebo díra (P vs. H)
- počáteční adresa
- délka



- **práce se seznamem:**

- proces skončí => P se nahrdí H
- dvě díry vedle sebe => sloučit do jedné
- seznam seřazený podle počáteční adresy oblasti

Alokace

- **first fit:**

- prohledávání od začátku dokud se nanajde dostatečně velká díra
- rychlý – prohledává co nejméně

- **next fit:**

- prohledávání začne tam, kde skončilo předchozí
- horší než first fit

- **best fit:**

- prohlédne celý seznam a vybere nejmenší díru kam se proces vejde

- pomalejší
- více ztracené paměti než FF, NF – zaplňuje paměť malými nepoužitelnými děrami
- **worst fit:**
 - největší díra
- **quick fit:**
 - samostané seznamy děr nejčastěji požadovaných délek (4KB, 8KB, ...)
 - ostatní velikosti v samostatném seznamu
 - alokace – rychlá
 - dealokace – obtížná
- **urychlení:**
 - oddělené seznamy pro proces a díry – pomalejší dealokace
 - oddělené seznamy, seznam děr dle velikost – optimalizace FF

Šetření pamětí

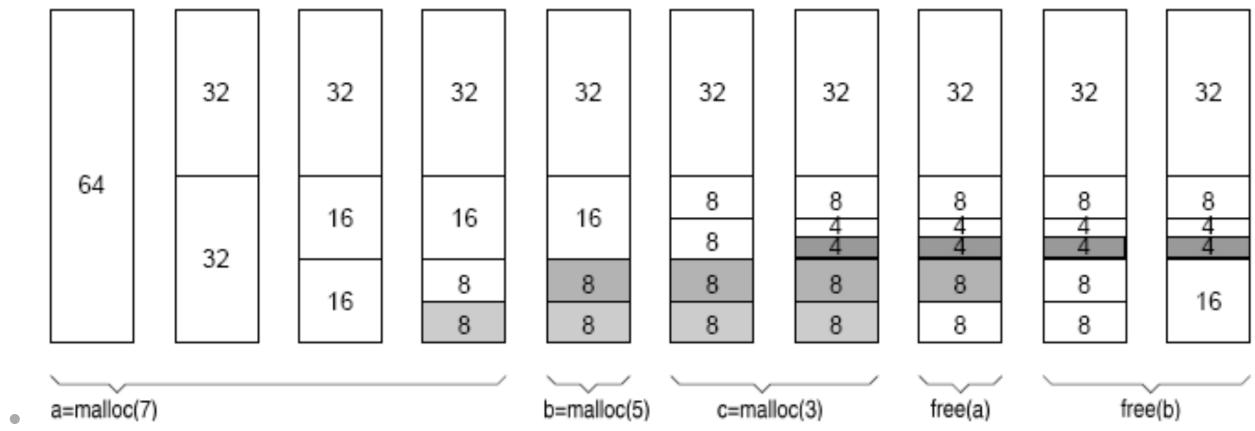
- díry využity k uložení seznamu děr
- **obsah díry:**
 - velikost díry
 - ukazatel na další díru

Asymetrie mezi procesy a dírami

- dvě sousední díry se sloučí
- dva procesy se nesloučí
- **při normálním běhu je počet děr poloviční oproti počtu procesů**

Buddy systems

- seznam volných bloků 1, 2, 4, 8, 16, ... alokačních jednotek až po velikost paměti
- nejprve seznamy prázdné vyjma 1 položky seznamu o velikosti paměti
- **příklad:**
 - alokační jednotka 1KB
 - paměť velikosti 64KB
 - seznamy: 1, 2, 4, 8, 16, 32, 64 (7 seznamů)
 - požadavek se zaokrouhlí na mocninu 2 (7KB se zakrouhlí na 8)
 - blok 64KB se rozdělí na 2 bloky 32KB (buddies) a dělíme dále a dále



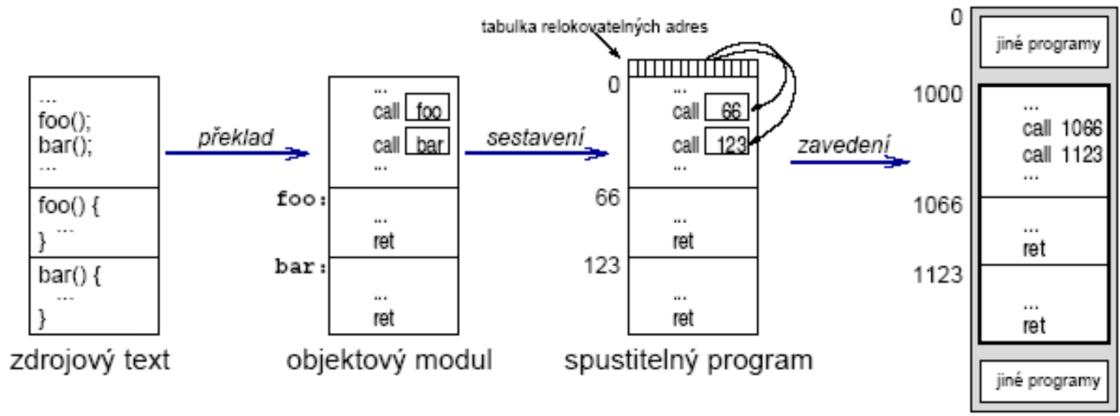
- neefektivní – plýtvá místem
- rychlý
- používá linux pro správu paměti jádra

Statická a dynamická relokace

- **relokace:**
 - programy běží na různých fyzických adresách
 - jednou je ve fyzické paměti od adresy 1000, jindy od 2000
- **ochrana:**
 - paměť musí být chráněna před zásahem jiných programů

Statická relokace

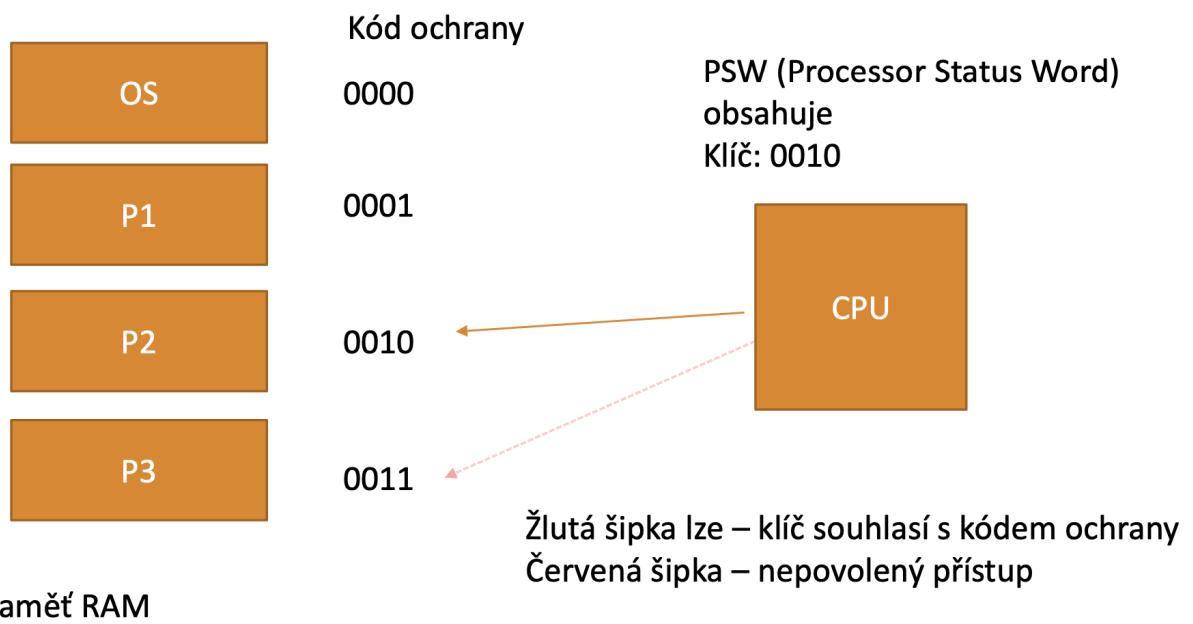
- **relokace při zavedení do paměti:**
 - **překlad a sestavení programu:**
 - **překlad:**
 - **objektové moduly** – výsledek překladu
 - příkazy přeloženy do strojových instrukcí
 - zůstávají symbolické odkazy – adresy proměnných, procedur, funkcí
 - **sestavení:**
 - symbolické odkazy jsou převedeny na číselné hodnoty
 - **statická realokace:**



- **statická relokace** = adresy se natvrdo přepíší správnými

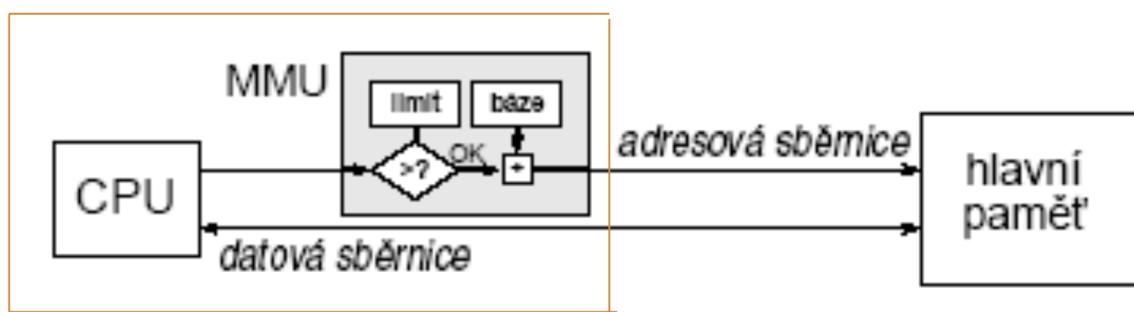
Mechanismy ochrany paměti

- **mechanismus přístupového klíče:**



- **mechanismus báze a limitu:**

- jednotka správy paměti MMU (uvnitř CPU)
- dva registry – **báze a limit**
- **báze** = počáteční adresa
- **limit** = velikost oblasti



- **funkce MMU** – převádí adresu používanou procesem na adresu do fyzické paměti
- tento mechanismus zajišťuje ochranu ale i dynamickou relokaci
-

Dynamická relokace

- provádí se za běhu
- bázi a limit může měnit pouze OS
- řeší **mechanismus báze a limitu**
 - můžeme zavést program od různé fyzické adresy (relokace)
 - můžeme zajistit aby proces nepřistupoval mimo svůj povolený rozsah paměti (ochrana)

Problém: procesy se nevezdou do paměti

- **řešení:**
 - **odkládání celých procesů (swapping):**
 - nadbytečný proces odložen na disk
 - **virtuální paměť:**
 - překrývání, stránkování, segmentace
 - v paměti nemusí být procesy celé

Správá paměti s odkládáním celých procesů

- data procesu mohou růst
- pro proces alokováno o něco více paměti, než je třeba
- potřeba více paměti než je alokováno:
 1. přesunout proces do větší oblasti (díry)
 2. překažející proces odložit
 3. dložit žadatele o paměť, dokud nebude prostor
 4. proces zrušit (odkládací paměť je plná)
- proces – dvě rostoucí části:
 - data, zásobník
 - rozrůstání proti sobě
 - překročení velikosti – přeúsn, odložit, zrušit
- **alokace odkládací oblasti:**

- jak vyhradit prostor pro proces na disku:
 - na celou dobu běhu programu (pořád do stejného místa)
 - alokace při každém odložení

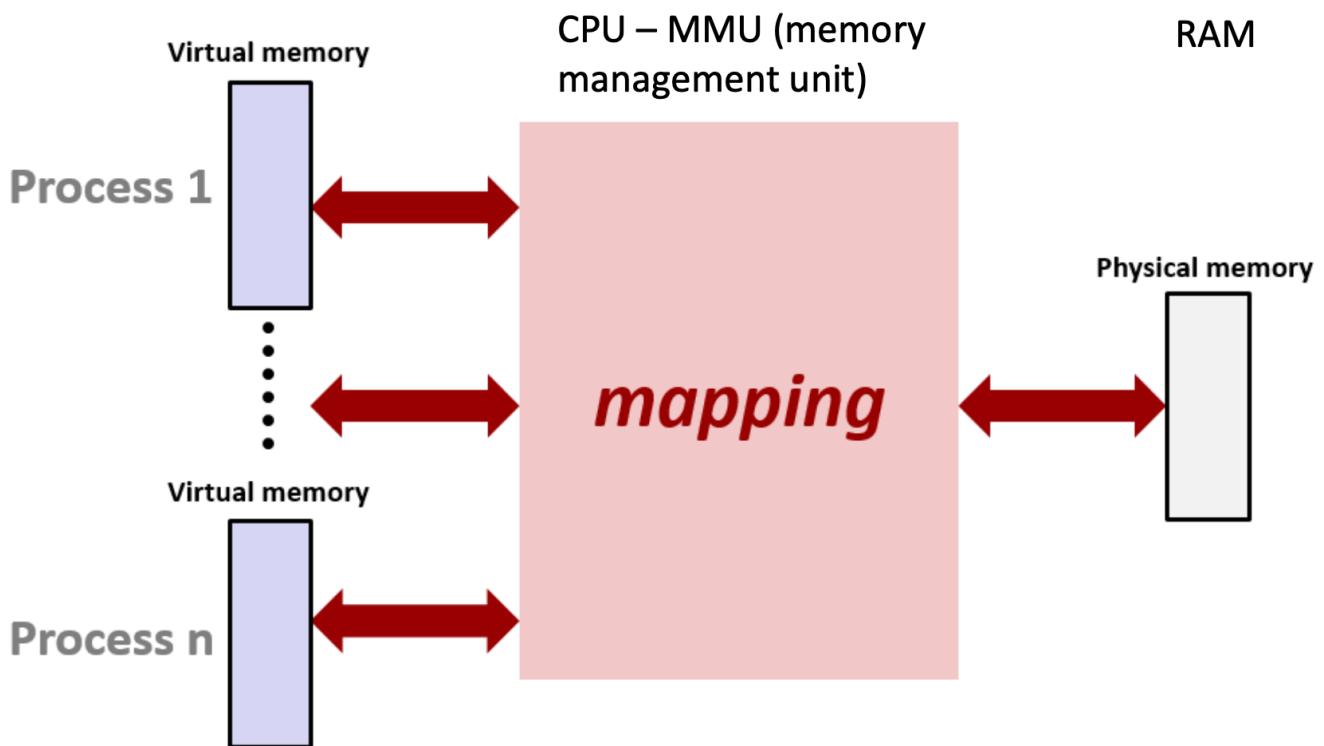
Virtuální paměť

- **mechanismus překrývání (overlays)** – starší, umožnil běh velkého programu
- **stránkování, segmentace** – používané dnes, nejčastěji stránkování

Překrývání (overlays)

- program rozdělen na moduly
- start – spuštěná část 0, při skončení zavede část 1, ...
- zavádění modů:
 - více překryvných modulů + data v paměti současně
 - moduly zaváděny podle potřeby
 - mechanismus odkládání
- zavádění modulů zařizuje OS
- rozdelení programů i dat na části – navrhuje programátor

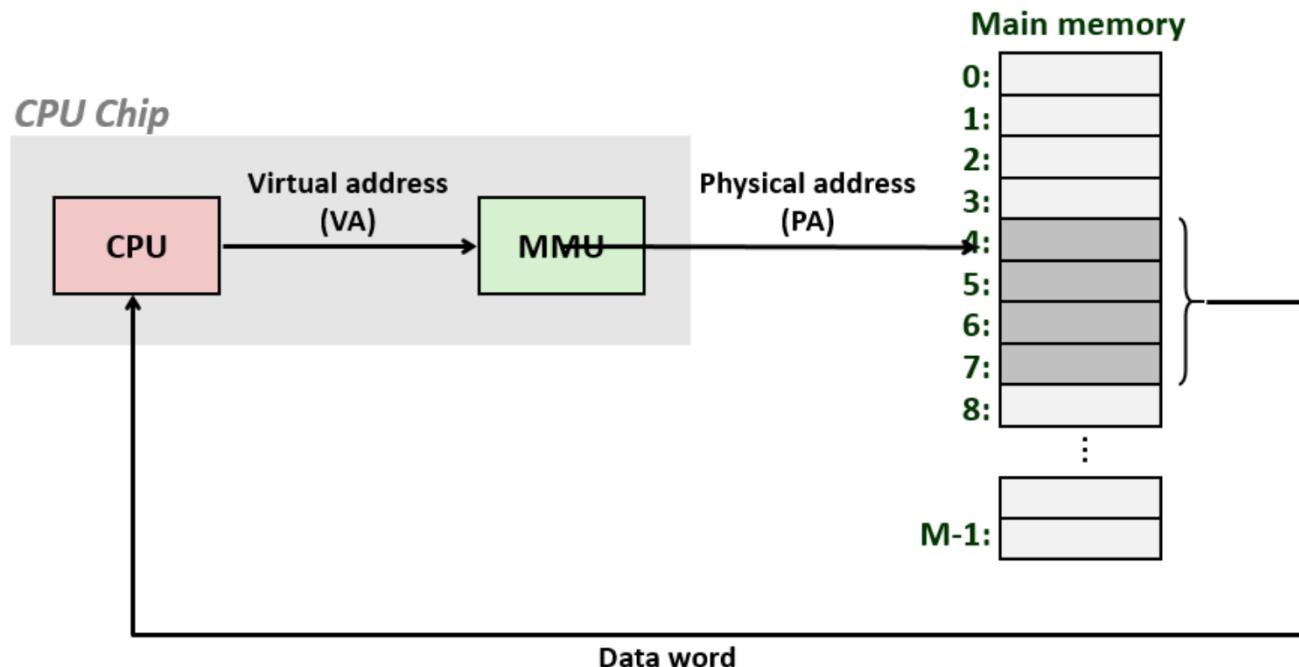
Virtuální adresy



- fyzická paměť RAM slouží jako cache virtuálního adresního prostoru procesů

- procesor – používá virtuální adresy u procesů
- pokud požadovaná část VA prostoru **JE** ve fyzické paměti – MMU převede VA => FA
- pokud požadovaná část VA prostoru **NENÍ** ve fyzické paměti – OS jí musí načíst z disku do RAM

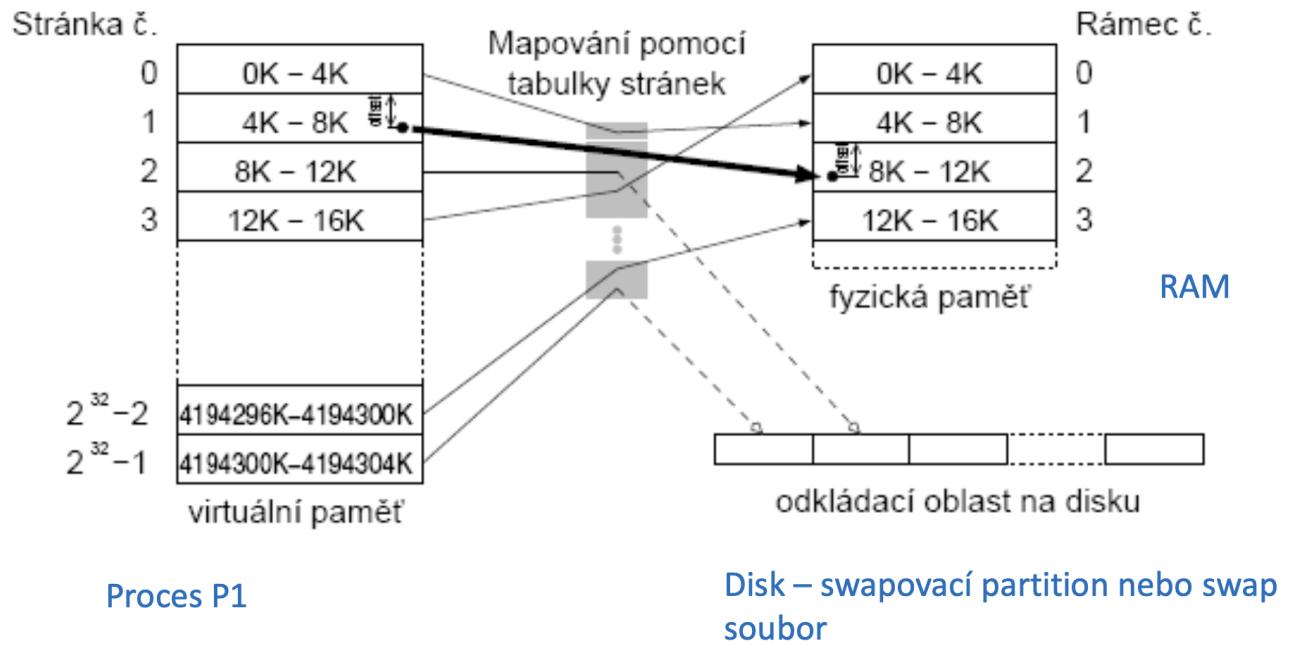
Vztah virtuální a fyzické adresy



Mechanismus stránkování (paging)

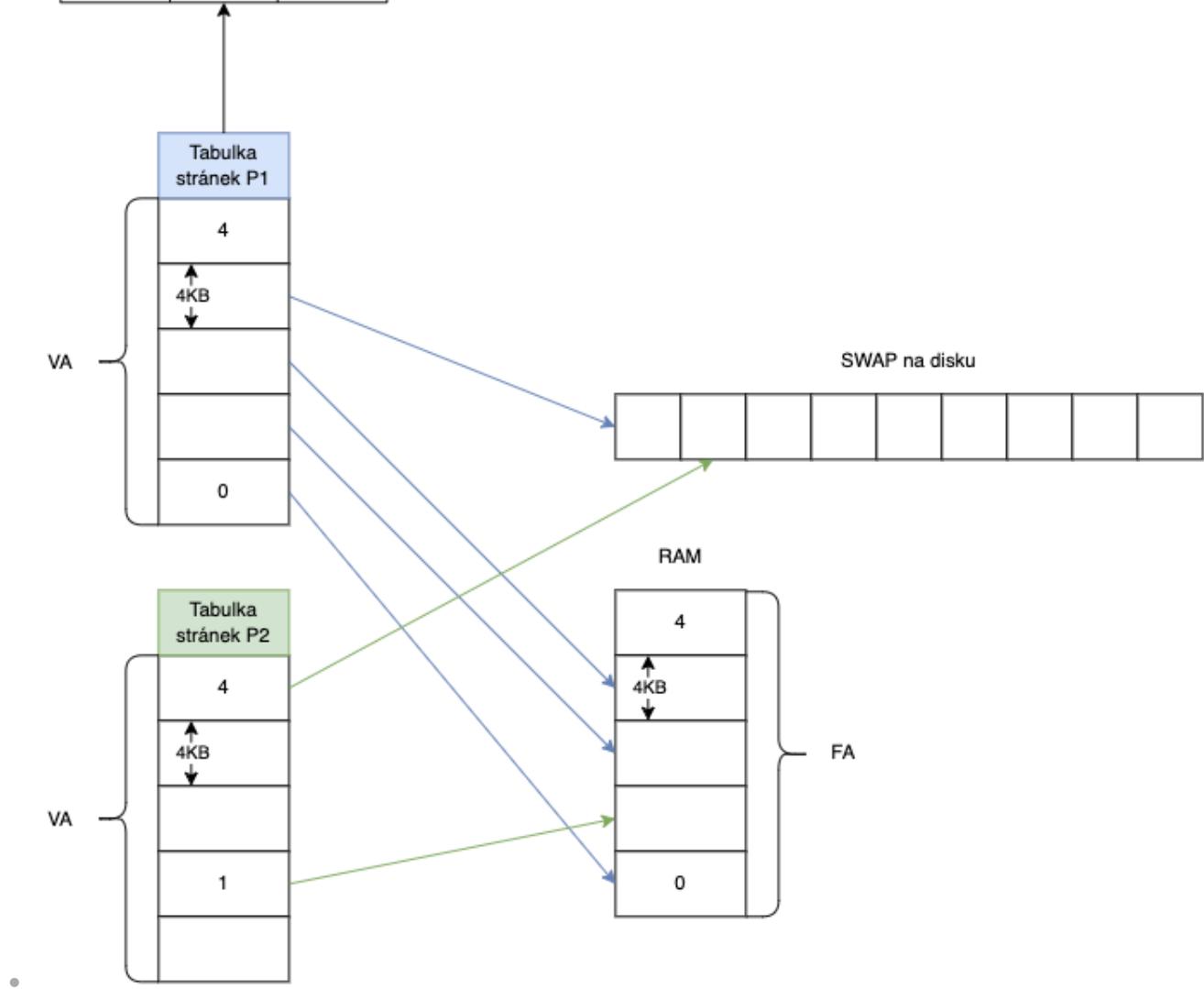
- **čisté stránkování** – bez swapu disku
- OS udržuje:
 - 1 tabulku rámců
 - tabulku stránek pro každý proces
- program využívá VA
 - **VA = číslo stránky + offset**
- musíme rychle zjistit zda je požadovaná adresa v paměti:
 - **ANO:** převod VA => FA
 - **NE:** zavedení z disku do RAM
- **VAP = stránky (pages)** pevné velikosti
- **fyzická paměť = rámce (page frames)** stejné velikosti jako stránky
- rámec může obsahovat **PRÁVĚ JEDNU** stránku
- na známém místě v paměti – **tabulka stránek** (register CR3 CPU ukazuje na tabulku)

- tabulka stránek mapuje VAP na rámce
- **dostupný virtuální adresní prostor:** $0 \dots 2^p - 1$ bytů (p – počet bitů ukazatele)



Příklad stránkovane paměti

stránka	rámec	další att.
0	0	
1	2	
2	3	
3	x	swap: 0
4		



- **přepočtení VA na FA pro proces 1:**

- VA = 4099
- číslo stránky = VA / velikost stránky = $4099 / 4096 = 1$
- offset = $4099 \bmod 4096 = 3$
- číslo rámce = tabulka_stranek_P1[1] = 2
- FA = číslo rámce * velikost stránky + offset

Tabulka stránek – podrobněji

Číslo stránky (index)	Číslo rámce	příznak platnosti	Příznaky ochrany	Bit modifikace (dirty)	Bit referenced	Adresa ve swapu
0	3	valid	rx	1	1	---
1	4	valid	rw	1	1	---
2	---	invalid	ro	0	0	4096

valid
invalid

rw, rx, ro,...

zda je třeba rámec uložit do swapu při odstranění z RAM

zda byla stránka přistupována (čtení či zápis) v poslední době

- **číslo rámce** = udává, v kterém rámci v RAM je stránka uložena
- **příznak platnosti** = říká zda je daná stránka v RAM nebo není
- **příznak ochrany** = zda je stránce jen pro čtení, nebo i pro zápis
- **bit modifikace** = pokud byla stránka modifikována (zápis), znamená to, že pokud je ve swapu, tak tam je nyní neaktuální
- **bit referenced** = zda byla stránka v nedávné době přístupována či ne
- velikost záznamů 32 bitů
- tabulka stránek je součástí PCB (Process Control Block)

Výpadek stránky

- stránka není mapována
- výpad způsobí vyjimku, zachycena OS (pomocí přerušení)
- OS inicializuje zavádění stránky a přepne na jiný proces
- po zavedení stránky OS upraví mapování (tabulkou stránek)

Problémy stránkování

- **velikost tabulky stránek:**
 - VA = 32 bitů
 - 12 bitů offset
 - 20 bitů číslo stránky
 - stránek je 2^{20}
 - každá položka zabírá 4B ... ($2^{20} * 4B$) = 4MB
 - proces využívá jen část prostoru VA
 - používá se víceúrovňová tabulka stránek
- **rychlosť převodu VA => FA:**

- každý přístup do paměti – sáhne do tabulky stránek
 - 2x více paměťových přístupů
- **TLB (Translation Look-aside Buffer)**
 - HW cache
 - **vstup:** číslo stránky
 - **výstup:** číslo_rámce nebo odpoví, že neví
 - **invertovaná tabulka stránek**

Ošetření výpadku stránky

1. výpadek – mechanismem přerušení vyvolán OS
2. OS zjistí, pro kterou stránku nastal výpadek
3. OS určí umístění stránky na disku
4. najde rámce, do kterého bude stránka zavedena
5. načte stránku do rámce (DMA přenos)
6. změní odpovídají mapovací položku v tabulce stránek
7. návrat
8. CPU provede instrukci, která způsobila výpadek

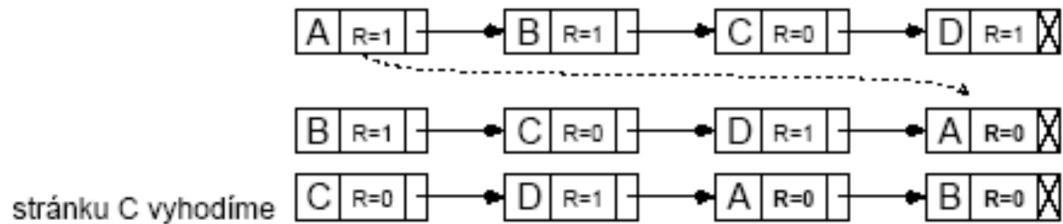
Algoritmy nahrazování stránky

- **FIFO:**
 - udržovat seznam stránek v pořadí ve kterém byly vytvořeny
 - využujeme nejstarší stránku
- **Beladyho anomálie:**
 - předpoklad: čím více bude rámců v paměti, tím nastane méně výpadků
 - Belady našel příklad pro algoritmus FIFO kdy to neplatí
- **MIN/OPT:**
 - optimální nejmenší možný výpadek stránek
 - stránka je označena počtem instrukcí, po který se k ní nebude přistupovat
 - vyřazuje se stránka s nejvyšším označením
 - pouze teoretické v praxi nefunguje
- **Least Recently Used (LRU):**
 - nejdéle nepoužítá
 - obtížná implementace
- **Not Recently Used (NRU):**
 - vyhazuje nepoužívané stránky
 - rozhoduje podle bitů **Reference (R)** a **Dirty (M = modified)**

- **4 kategorie:**

- třída 0: R = 0, M = 0
- třída 1: R = 0, M = 1
- třída 2: R = 1, M = 0
- třída 3: R = 1, M = 1
- vyhodí stránku z nejnižší neprázdné třídy

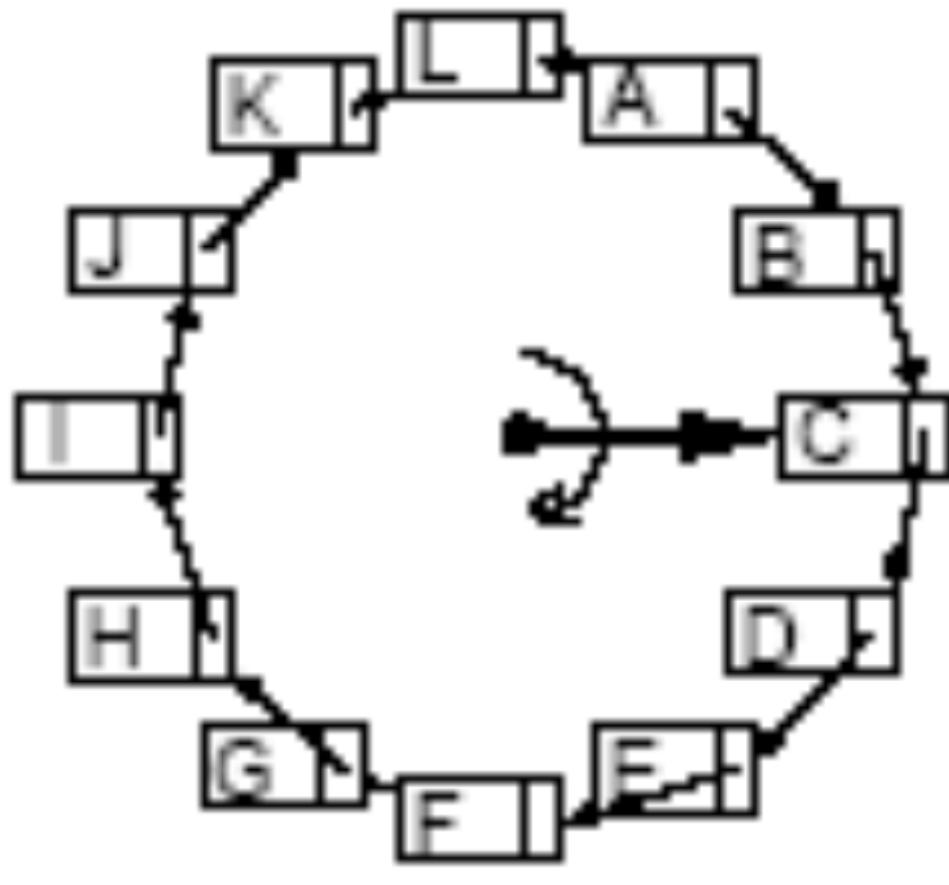
- **Second Chance:**



1. Krok – nejstarší je A, má R = 1 – nastavíme R na 0 a přesuneme na konec seznamu
2. Druhá nejstarší je B, má R = 1 – nastavíme R na 0 a opět přesuneme na konec seznamu
3. Další nejstarší je C, R = 0 – vyhodíme ji

- **Clock:**

- jako Second Chance ale v kruhovém seznamu
- stránka kam ukazuje ručička => pokud R=0 vyhodit, pokud R=1 nastavit na 0 a posun



Alokace fyzických rámců

- **globální** – pro vyhození se uvažují všechny rámce
 - lepší průchodnost systému
 - ne běh procesu má vliv chování ostatních procesů
- **lokální** – uvažují se pouze rámce alokované procesem
 - počet stránek alokovaných pro proces se nemění
 - program se chová stejně při každém běhu
- **typy:**
 - **nejjednodušší** – všem procesům dát stejeně
 - **proporcionální** – každému proporcionalní díl podle velikosti procesu
 - **nejlepší** – podle frekvence výpadků stránek za jednotku času **Page Fault Frequency (PFF)**

Zloděj stránek (page deamon)

- v systému se udržuje určitý počet volných rámci
- když klesne pod určitou mez pustí **page deamon - kswapd**, ten uvolní určité množství stránek (rámců)
- **mlock** – zamykání stránek

Zahlcení a pracovní množina stránek

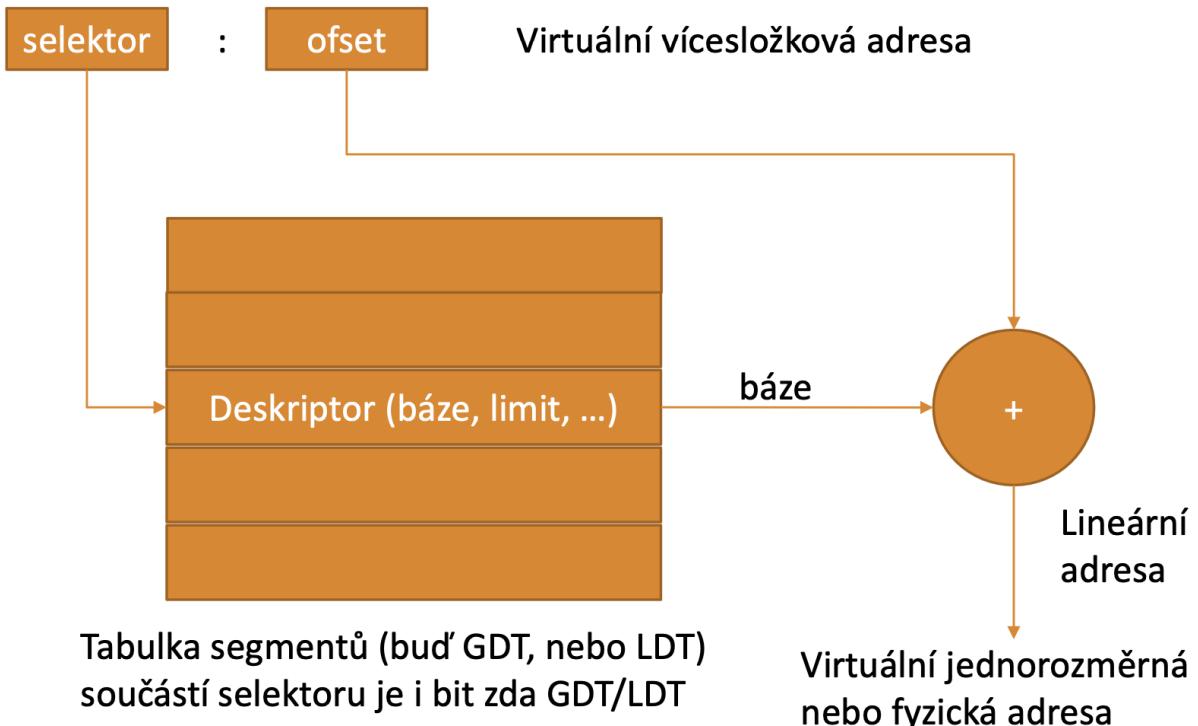
- proces pro svůj rozumný běh potřebuje **pracovní množinu stránek**
- pokud se pracovní množiny stránek aktivních procesů nevejdou do paměti, nastane zahlcení = thrashing
- **zahlcení:**
 - v procesu nastane výpadek stránky
 - paměť je plná (není volný rámec) – je třeba nějakou stránku vyhodit
 - stránka pro vyhození bude ale brzo potřebná
 - bude se muset vyhodit jiná používaná stránka
 - uživatel zahlcení pozoruje – běh procesů se zpomalí
 - **řešení** – při zahlcení snížit úroveň multiprogramování

Mechanismus segmentace

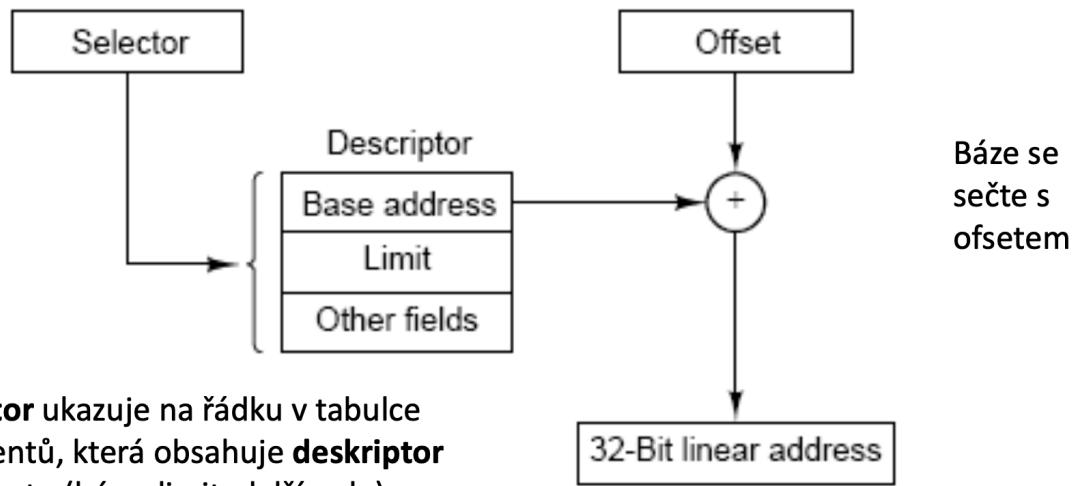
- více samostatných virtuálních adresových prostorů
- **segment** = logické seskupení informací
 - lineární posloupnost adres od 0
- každý segment – logická entita → má samostatnou ochranu

Čistá segmentace

- každý odkaz do paměti dvojice (**selektor, offset**)
 - **selektor** – obsahuje odkaz do tabulky segmentů, určuje segment
 - **offset** – relativní adresa v rámci segmentu
- nutnost přemapování dvojice (selektor, segment) na lineární adresu (je již fyzická, když není dále stránkování)
- **atributy položky tabulky segmentů:**
 - počáteční adresu segmentu = **báze**
 - rozsah segmentu = **limit**
 - příznaky ochrany segmentu (rwx)
- **tabulka segmentů obsahuje:** deskriptory segmentů
- **GDT (Globální tabulka deskriptorů segmentu)** – jedna pro celý systém
- **LDT (Lokální tabulka deskriptorů segmentu)** – každý proces může mít vlastní
- mezi GDT/LDT se volí jedním bitem
- systém musí být schopen najít kde leží LDT procesu
- **selektor, offset, deskriptor:**



- Tabulka segmentů (buď GDT, nebo LDT) součástí selektoru je i bit zda GDT/LDT
- **kontroly:**
 - ukazuje selektor na vyplněnou řádku v DT? – pokud ne konec procesu
 - platí, že offset \leq limit? – pokud ne konec procesu
 - nesahám na segment kam může jen jádro? – pokud sahám konec procesu
- **(selector, offset) => lineární adresa:**



- **převod na fyzickou adresu:**
 - PCB obshuji odkaz na tabulku segmentů procesu
 - odkaz do paměti má tvar (selector: offset)
 - možnost sdílet segment mezi více procesy

- **instrukce: LD R, sel:offset**

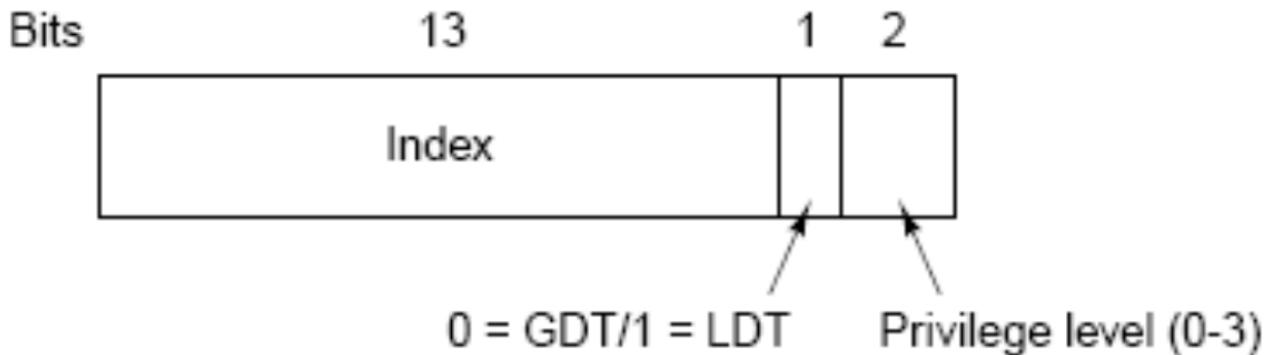
1. selektor – obsahuje deskriptor segmentu
2. kontrola offset < limit, ne – porušení ochrany paměti
3. kontrola zda dovolený způsob použití, ne – chyba
4. adresa = báze + offset

Segmentace se stránkováním

- velké segment – nepraktické celé udržovat v paměti
- stránkování segmentů – v paměti pouze potřebné stránky
- **adresy:**
 - **virtuální adresa => lineární adresa => fyzická adresa**
 - **virtuální** – používá proces (selektor:offset)
 - **lineární** – po segmentaci, pokud není stránkování představuje fyzickou adresu
 - **fyzická** – adresa do fyzické paměti RAM
- **segmentové registry:**
 - **CS (Code Segment)**
 - **DS (Data Segment)**
 - **SS (Stack Segment)**

Selektor segmentu

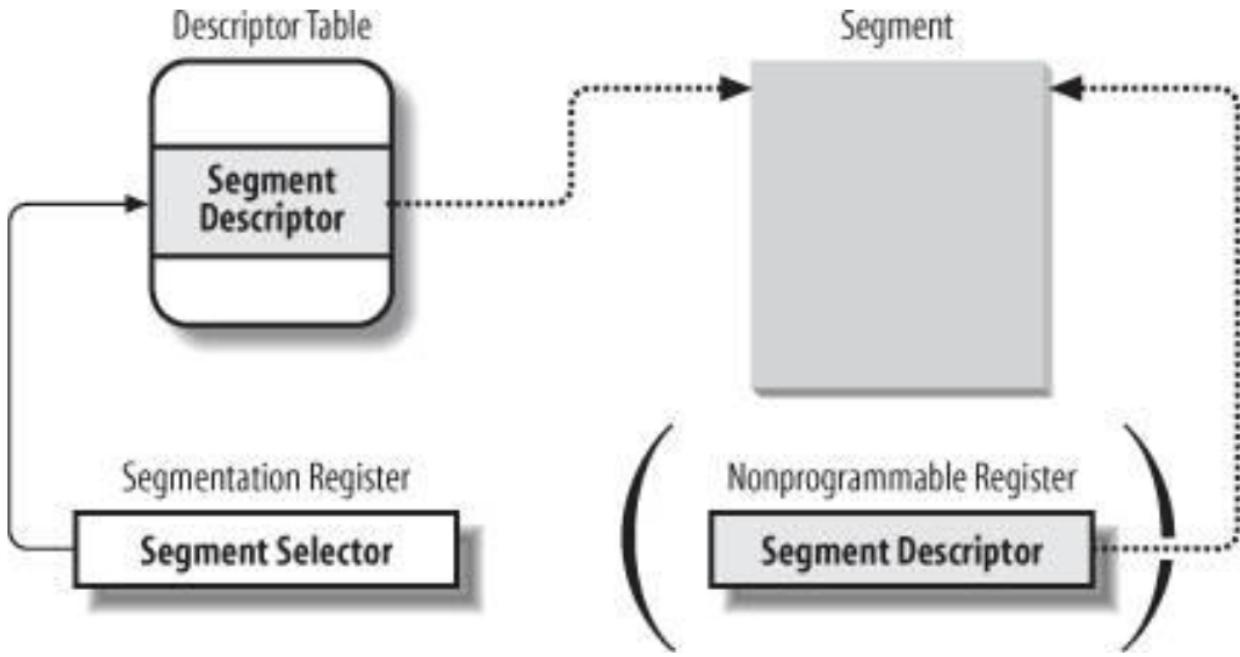
- selektor – 16bitový
- 13bitů – index do GDT nebo LDT
- 1 bit – 0 = GDT, 1 = LDT
- 2 bity – úroveň privilegovanosti (0-3, 0 – jádro, 3 – uživatelské procesy)



- selektor 0 – nedostupnost segmentu
- maximálně $2^{13} = 8192$ položek

Rychlý přístup k deskriptoru segmentu

- **logická adresa:** segment selektor (16bitů) + offset (32bitů)
- **zrychlení převodu:** přídavné neprogramovatelné registry



Deskriptor segmentu

- **64bitů:**
 - **32 bitů** = báze
 - **20 bitů** = limit
 - v bytech do 1MB (2^{20})
 - v 4K stránkách do $2^{12} = 4096$
 - **příznaky**
 - typ a ochrana segmentu

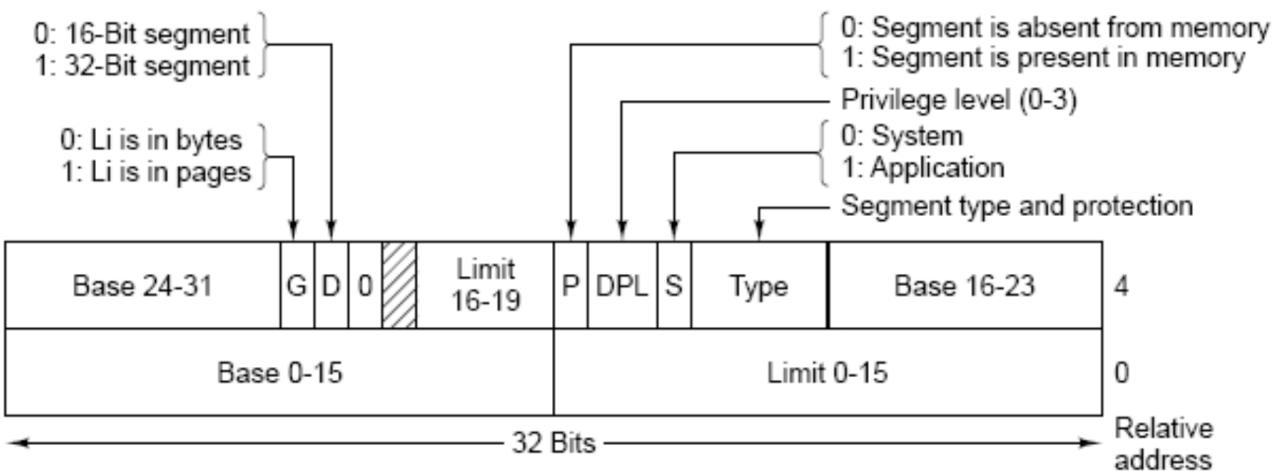
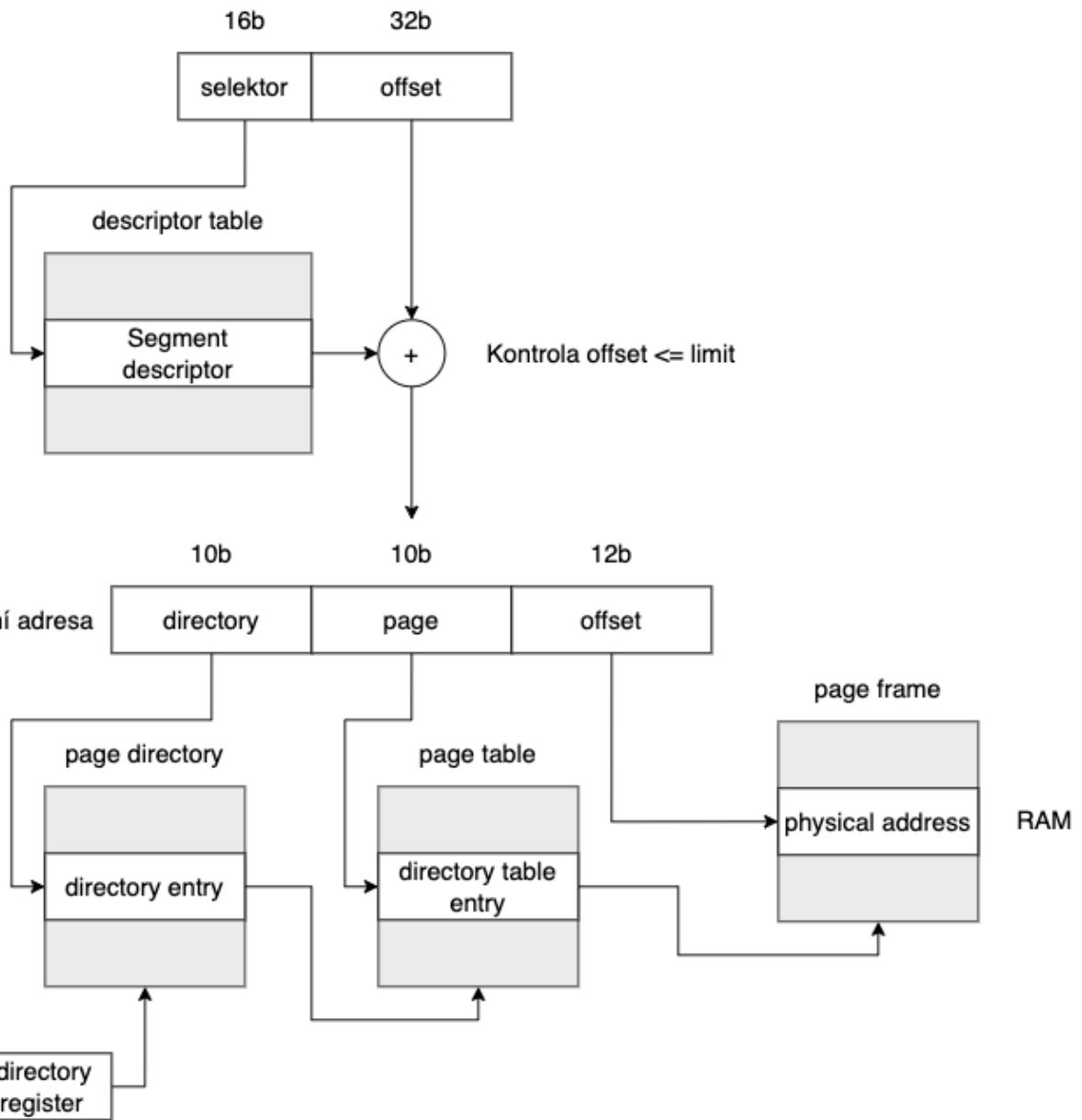


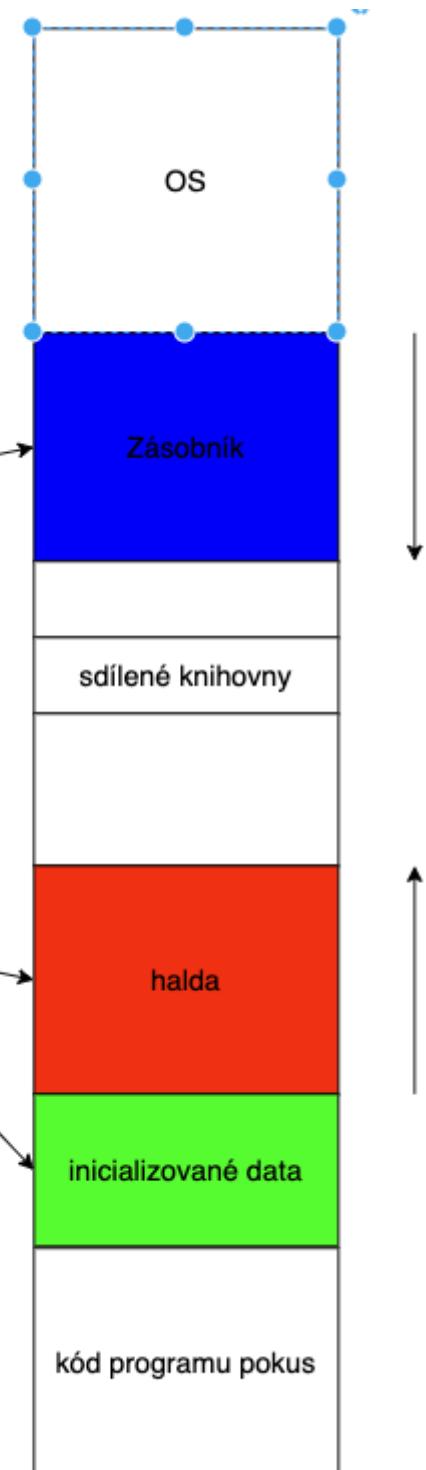
Schéma převodu VA na FA



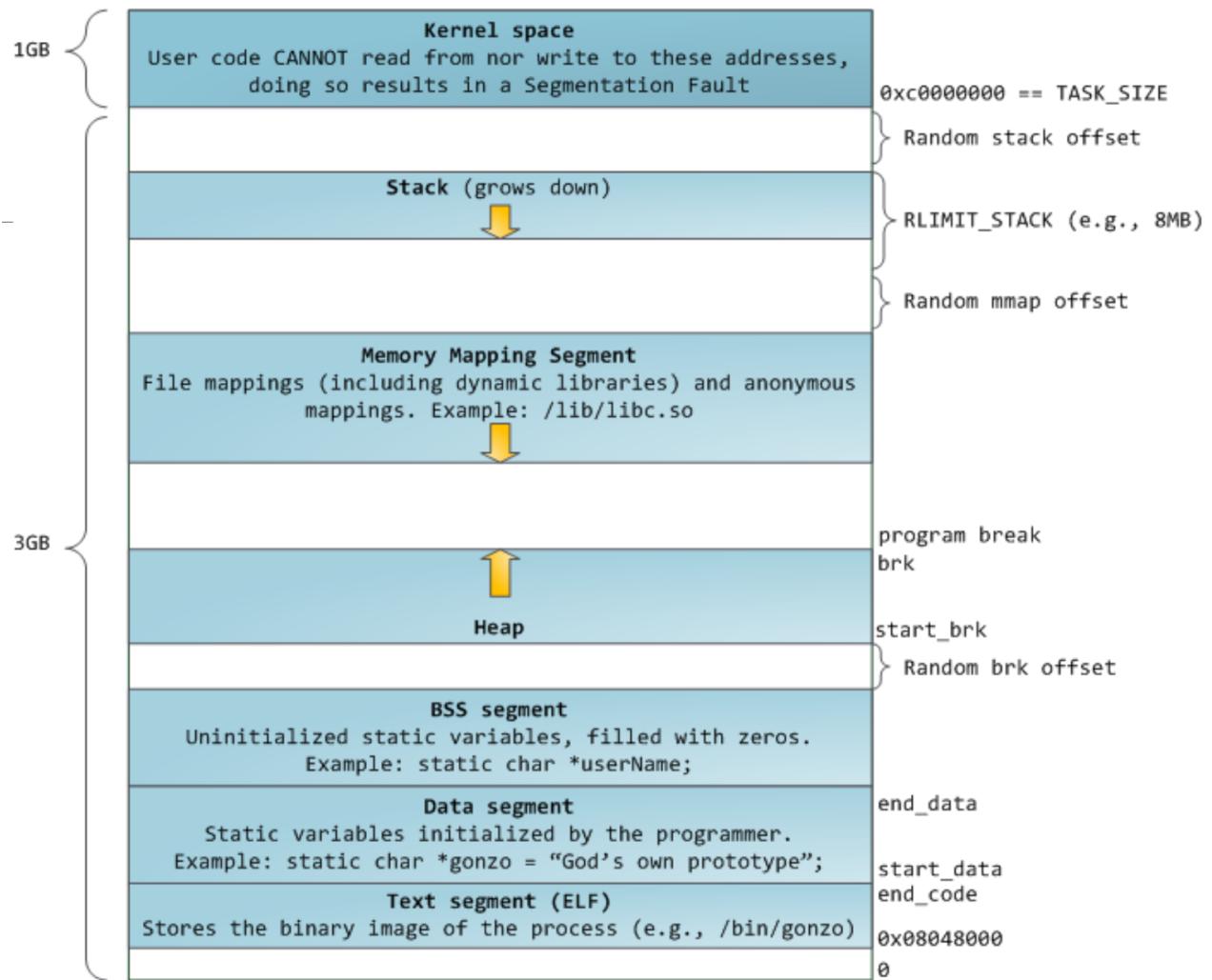
Rozdělení paměti pro proces

pokus.c:

```
int x = 5; int y = 7;           // inicializovaná data
void fce1() {
    int pom1, pom2;           // na zásobníku
    ...
}
int main (void) {
    ...
    ptr = malloc(1000);       // halda
    fce1();
    return 0;
}
```

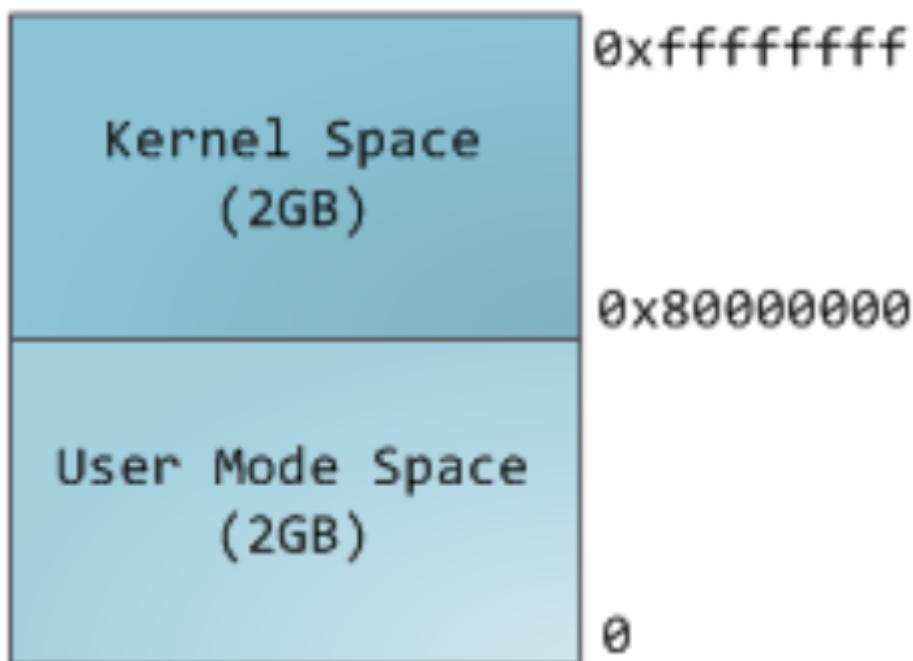


- 3 + 1:

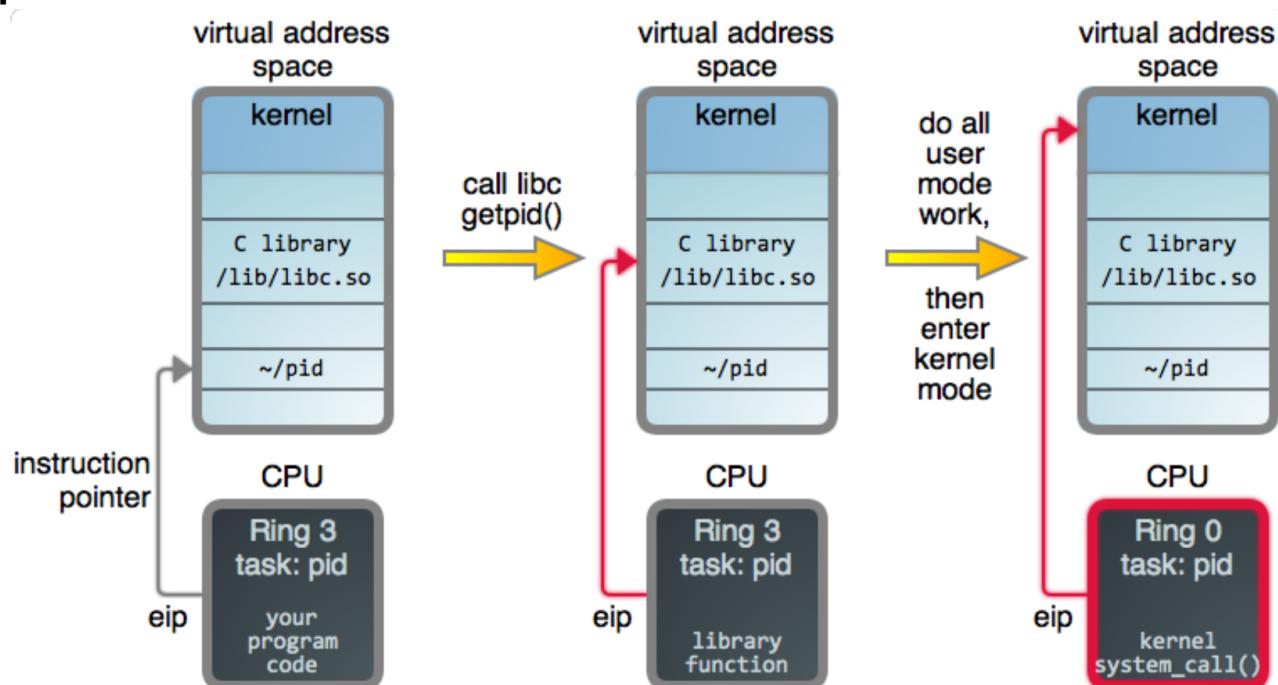


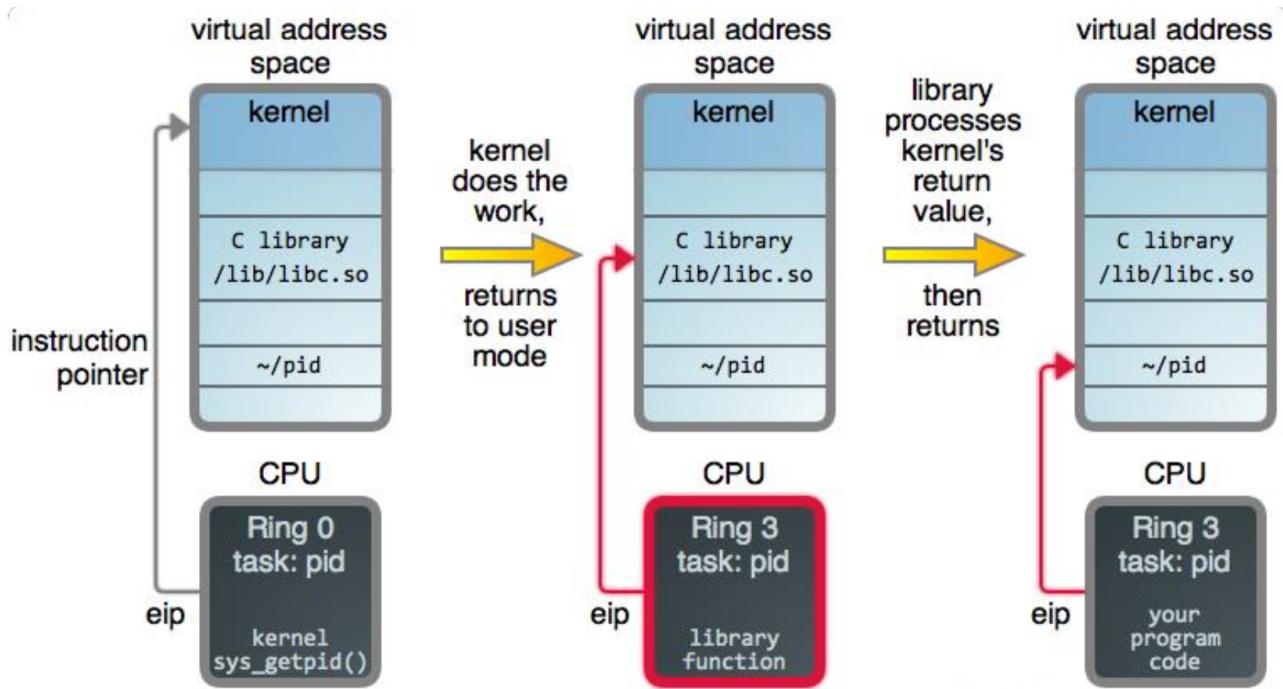
- 2 + 1:

Windows, default memory split



- příklad:





Vnější a vnitřní fragmentace

- **vnější:**
 - zůstávají nepřidělené úseky paměti
 - např. dynamické přidělování
 - při stránkování nenastává
- **vnitřní:**
 - část přidělená oblasti je nevyužita (dostaneme přidělenou stránku, ale využijeme jen část)
 - při stránkování nastává – v průměru polovina poslední stránky procesu je prázdná

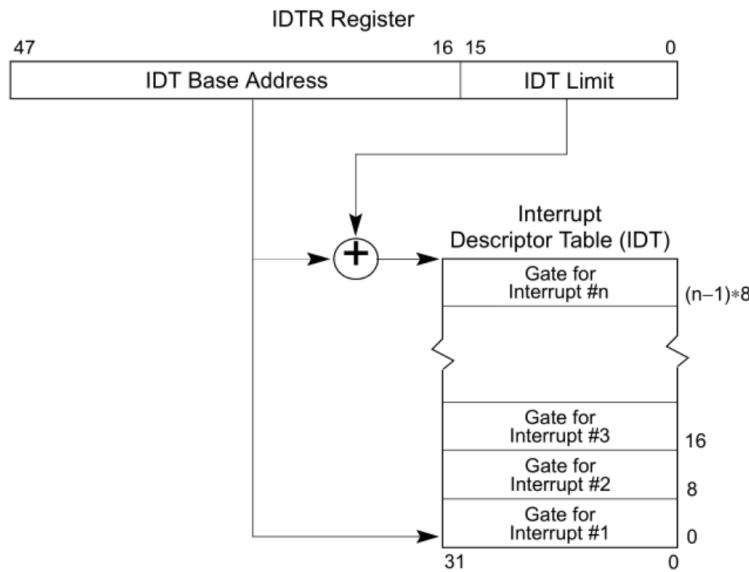
Procesory x86

- **real mode (MS-DOS)**
 - po zapnutí napájení, žádná ochrana paměti
 - FA = segment * 16 + offset
 - FA 20bitová, segment, offset .. 16 bitové
- **protected mode (dnešní OS)**
 - nastavíme tabulkou deskriptorů (min. 0, kód, data)

- a nastavíme PE bit v CRO registru

Procesory a přerušení

- **reálný mód:**
 - první KB (0...1023) RAM – interrupt vector table
- **chráněný mód:**
 - IDT (Interrupt Descriptor Table)
 - pole 8bytových deskriptorů
 - naplněná IDT tabulka 2KB (256x8B)
 - umístění tabulky je v registru IDTR
- **IDTR registr:**



Určuje, kde
tabulka vektorů
přerušení začíná
(báze)

i její velikost
Báze+limit =konec
tabulky

- **call gates** – volání předdefinované funkce přes `CALL FAR selector_callgate`
- **task state segment** – přepínání tasků

Chráněný režim – adresy

- **VA(selektor,offset)=segmentace => LA=stránkování => FA**
- segmentaci nejde vypnout, stránkování ano
 - bit v řídícím registru CPU CRO – udává zda je zapnuté stránkování
- je-li vypnuto stránkování $LA = FA$

- chce-li systém používat jen stránkování roztáhne segmenty přes celý adresní prostor

Alokace paměti pro procesy

- **explicitní zpráva paměti** – programátor se stará o uvolnění
- **čítání referencí** – každý objekt má u sebe čítač referencí
- **garbagr collection** – pokročilé algoritmy

Čítání referencí

- každému objektu přiřazen čítač referencí
- někdo si uloží referenci na objekt – zvýšení čítače
- reference mimo rozsah platnosti nebo přiřazená nová hodnota jinam – snížení čítače
- čítač na nule – uvolnění objektu z paměti
- nevýhoda cyklus – dva ukazují na sebe ale nic dalšího na ně
- nevýhoda – režie

Garbage collection (GC)

- automatická správa paměti
- speciální alg. hledá a uvolňuje úseky paměti, které již proces nevyužívá
- zjišťování, které objekty jsou z kořene programu nedostupné
- **sledovací algoritmy:**
 - přeruší běh programu a vyhledávají dosažitelné objekty
 - **algoritmus Mark and Sweep:**
 - všechny objekty hodnota navštíven FALSE
 - projde všechny objekty
 - ty, které navštívý nastaví je navštíven na TRUE
 - objekty s FALSE smaže
 - nevýhody: přerušení běhu programu, když nenechává živé objekty na místě – fragmentace paměti
- **kopírovací algoritmus:**
 - rozdělí haldu na dvě části (aktivní a neaktivní)
 - pokud se při alokaci nevezde do dané části haldy provede se úklid

- úklid – prohození aktivní a neaktivní části, do aktivní se kopírují živé objekty ze staré
- 2x velikost haldy, potřeba kopírovat objekty, které přežijí úklid
- **generační algoritmy:**
 - paměť do několika generací (částí)
 - objekty vytvářeny v nejmladší
 - po dosažení stáří přesun do starší generace
 - pro každou generaci – úklid v různých časových intervalech
 - nejmladší generace se zaplní – všechny dosažitelné v nejmladší zkopirovány do starší

Amdahlův zákon

- určuje urychlení výpočtu při užití více procesorů
- urychlení je limitováno sekvenčními částmi výpočtu

Škodlivé programy

- **bacteria** – konzumuje systémové zdroje tím, že se replikuje
- **logic bomb** – pokud v systému nastanou nějaké podmínky provede se nějaká škodlivá akce
- **trapdoor** – tajný nedokumentovaný vstupní bod do programu
- **trojan horse** – tajná nedokumentovaná rutina obsažená v jinak užitečném programu
- **virus** – kód zapouzdřený v programu, kopíruje se a vkládá do dalších programů
- **worm** – program může replikovat sebe a šířit se z PC na PC přes síť
- **ransomware** – zablokuje PC nebo šifruje zapsaná data a pak požaduje od oběti výkupné