Operační systémy

Přednáška 1: Úvod

Organizace předmětu

- **Přednášky**
  - každé úterý 18:00-19:30 v D209
- **Přednášející**
  - Jan Trdlička
  - email: trdlicka@fel.cvut.cz
  - kancelář: K324
- **Cvičení**
  - pondělí, úterý, středa, pátek
- **Informace o předmětu** (přednášky, cvičení, zkouška,…)

  http://service.felk.cvut.cz/courses/X36OSY

Podmínky pro získání zápočtu/zkoušky

- Semestrální úloha – vlákna (max. 30 bodů)
- Semestrální úloha – procesy (max. 30 bodů)
- Zkouškový test (max. 40 bodů)

- **Nutnou podmínkou pro zápočet**
  - aspoň 10 bodů z každé úlohy
  - pozdní odevzdání úlohy = 0 bodů

- **Nutnou podmínkou pro zkoušku**
  - zápočet + aspoň 10 bodů ze zkouškového testu

- **Klasifikace**

  55 < dobře ≤ 70 < velmi dobře ≤ 85 < výborně

O předmětu

- **Cíl předmětu**
  - pochopit základní principy používané v operačních systémech (OS)

- **Obsah předmětu**
  - definice, úkoly, typy, struktura, historie OS
  - procesy a vlákna, výměna informací, synchronizace
  - přidělování prostředků, uváznutí
  - virtuální paměť, stránkování, segmentace, náhrada stránek
  - uložení dat, RAID, systémy souborů
  - principy V/V, přerušení, DMA
  - sítové rozhraní, TCP/IP
  - bezpečnost

Literatura

– Tanenbaum, A.: *Modern Operating Systems*, 2nd Ed., Prentice Hall, ISBN 0-13-031358-0, 2001.

– Stallings, W.: *Operating Systems*, 5th Ed., Prentice Hall, ISBN 0-13-147954-7, 2005.

– Silberschatz, A.: *Operating System Concepts,* 7th Ed., John Wiley, ISBN 0-471-69466-5, 2005.

– Plášil, F., Staudek, J.: *Operační systémy*, Praha, SNTL, 1992.

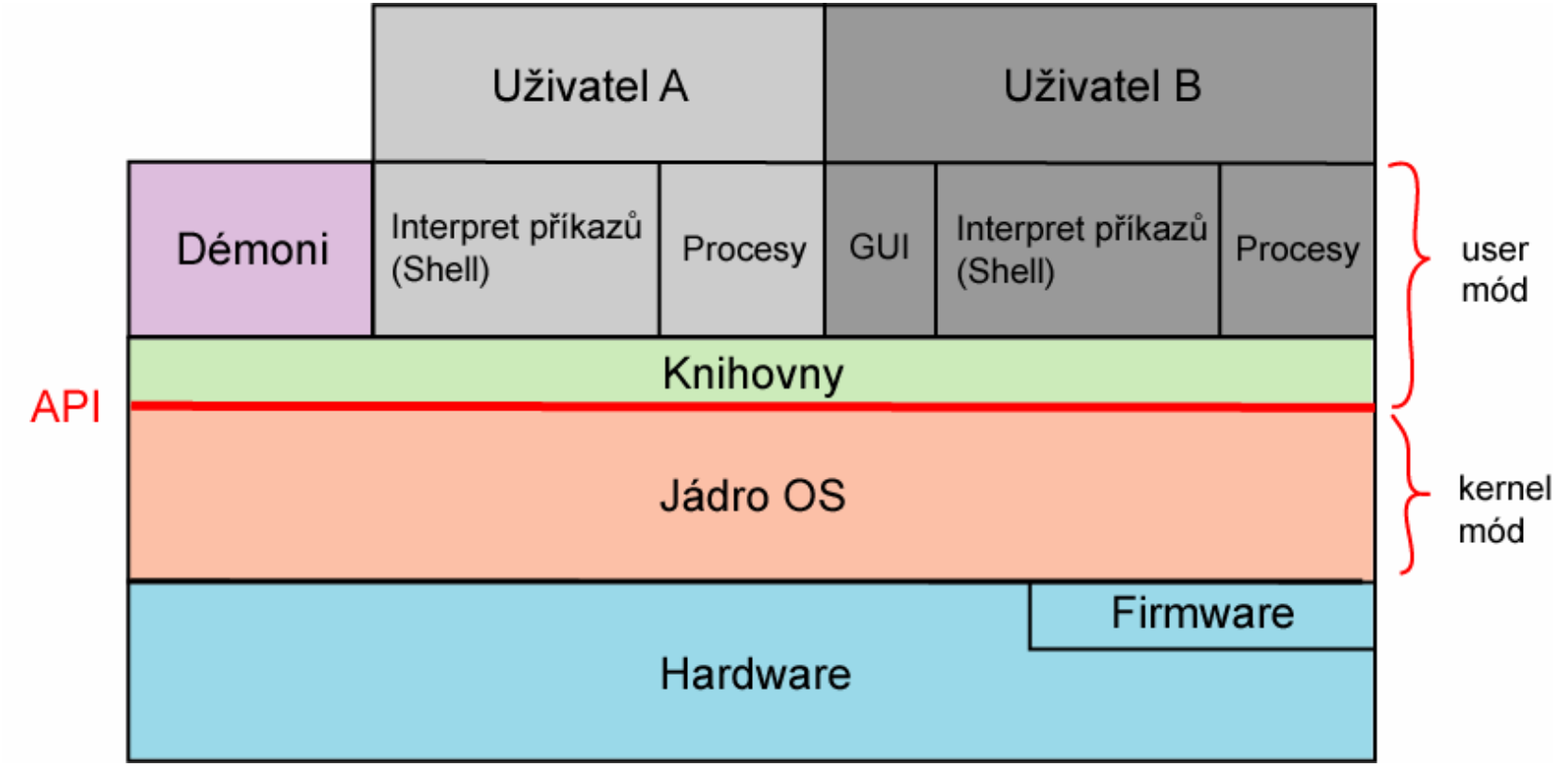
Operační systém (OS)

- **Definice**
  - základní SW, který funguje jako prostředník mezi HW a aplikacemi/uživateli

- **Úkoly**
  - **správa výpočetních prostředků**
    - fyzických (procesor, paměť, disky, …)
    - logických (uživatelská konta, procesy, soubory, přístup. práva,…)
  - **abstrakce složitosti HW**
  - **poskytuje rozhraní**
    - aplikacím (Win32 API, Win64 API, systémová volání Unixu,…)
    - uživatelům (CLI a GUI)

- Soustředíme na **principy univerzálních OS**
  - MS Windows, Unix OS, VMS, Mac OS, …

Výpočetní systém



Požadavky na HW

- **podpora OS při správě HW**
  - CPU s dvěma módy
  - **kernel mód**: CPU může provádět všechny instrukce ze své instrukční množiny (program běžící v tomto módu má přístup k celému hardwaru)
  - **user mód:** pouze podmnožina instrukcí je povolena (např. všechny instrukce V/V a ochrany paměti jsou zakázány)

- **podpora OS při správě paměti**
  - ochrana paměti
  - podpora virtuální paměti (fyzický x logický prostor)
  - překlad LA na FA

Požadavky na HW (2)

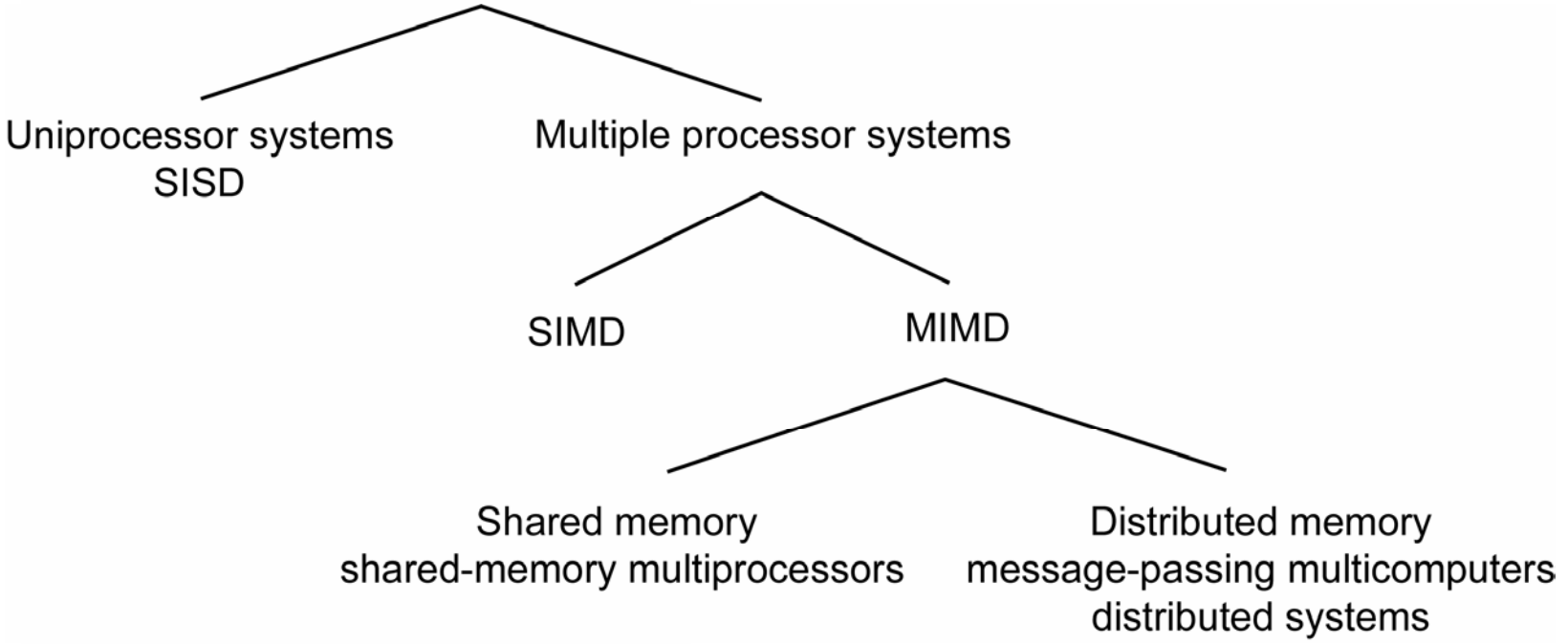
- **podpora přepínání kontextu**
  - mechanismus přerušení (reakce na asynchronní události)
  - programovatelný interní časovač (generuje přerušení po uplynutí čas. kvanta)

- **podpora efektivních operací V/V**
  - přerušení, DMA, …

Flynova klasifikace HW

- **Single Instruction Single Data stream (SISD)**
  - 1 procesor provádí 1 instrukční proud nad daty uloženými v 1 paměti
- **Single Instruction Multiple Data stream (SIMD)**
  - 1 instrukce je prováděna nad množinou dat množinou procesorů
  - vektorové a maticové počítače
- **Multiple Instruction Single Data stream (MISD)**
  - posloupnost dat je přenášena k množině procesorů provádějících různé instrukce
  - nebylo implementováno
- **Multiple Instruction Multiple Data stream (MIMD)**
  - množina procesorů současně provádí různé instrukce nad různými daty
  - multiprocesory, multipočítače, distribuované systémy

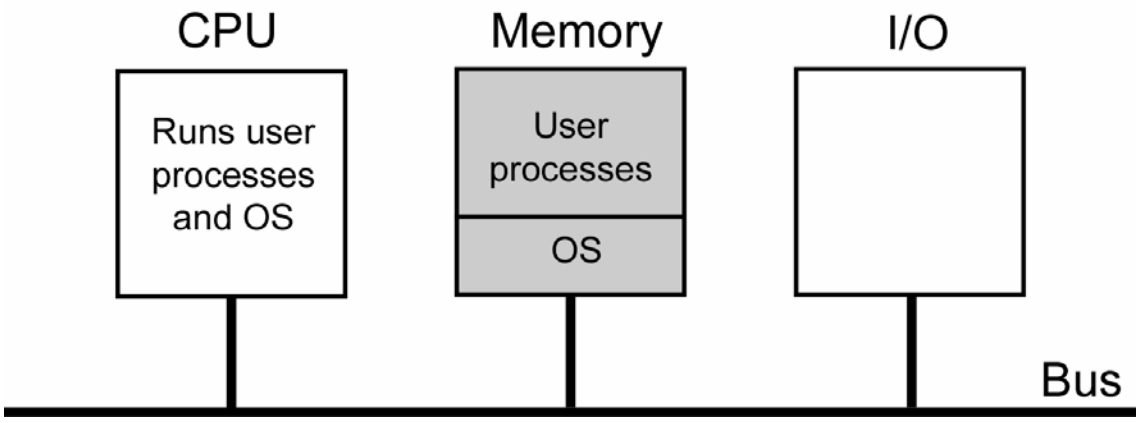
Architektura paralelních systémů



Jednoprocesorové systémy (SISD)

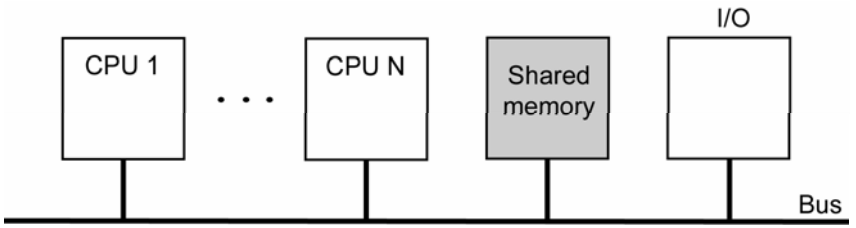
- CPU načítá instrukce a data z paměti, zpracovává je a výsledky zapisuje zpět do paměti

- Paměť obsahuje jádro OS + další procesy

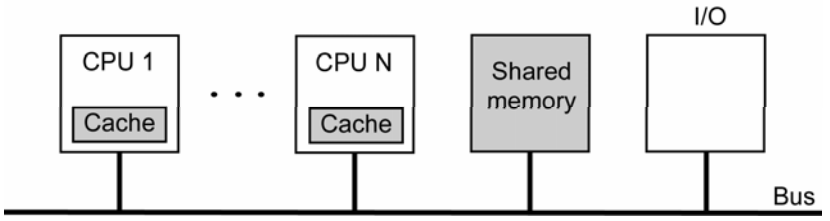


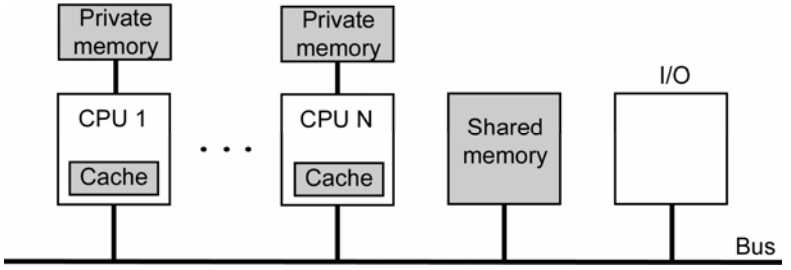
Multiprocesory se sdílenou pamětí

- Komunikace mezi CPU přes sdílenou paměť (10-50ns)
- **UMA (Uniform Memory Access) multiprocesory**
  - přístup do sdílené paměti je stejně rychlý pro všechny CPU



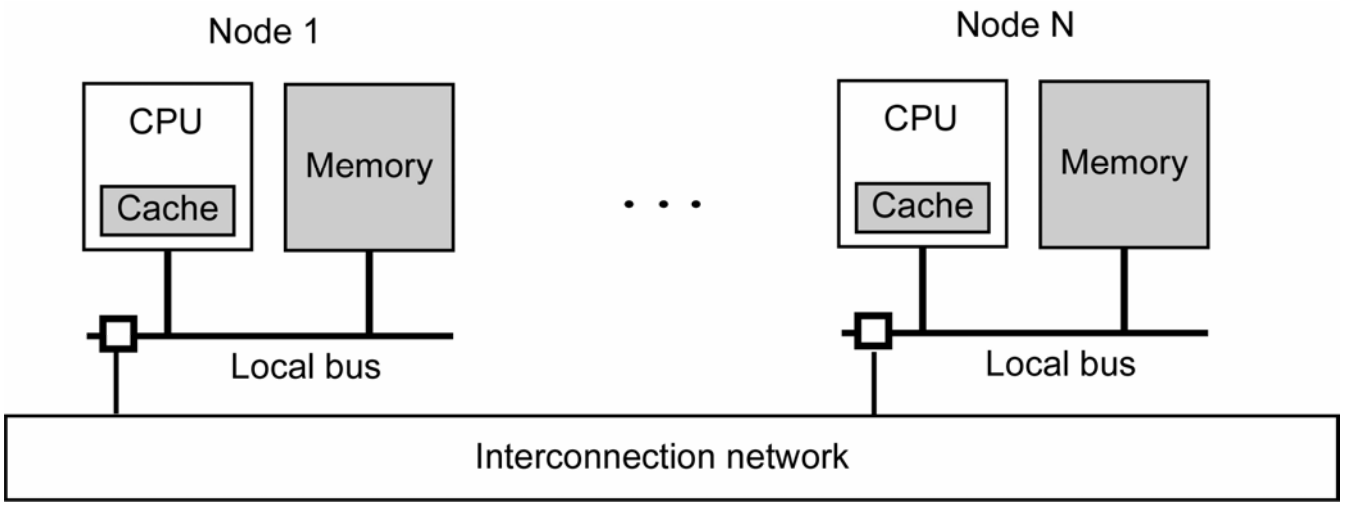
  - sběrnice je slabé místo
    - použití vyrovnávacích pamětí a lokální paměti
    - místo paměťové sběrnice speciální přepínač (crossbar switch)





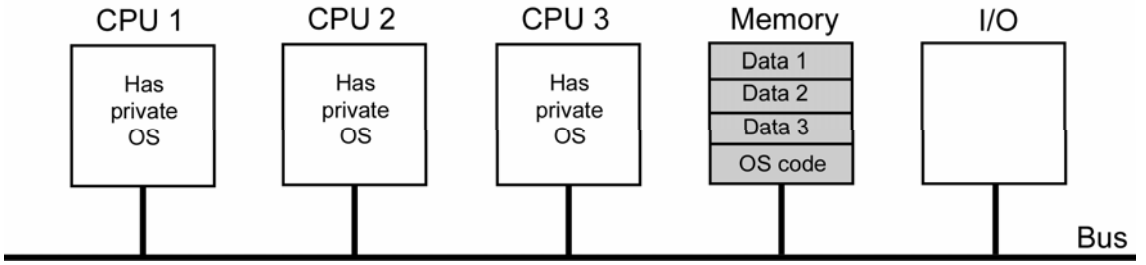
Multiprocesory se sdílenou pamětí (2)

- **NUMA (Nonuniform Memory Access) multiprocesory**
  - lokální paměť je viditelná pro ostatní CPU
  - přístup do vzdálené paměti pomocí LOAD a STORE instrukcí
  - přístup do vzdálené paměti je pomalejší než do lokální paměti

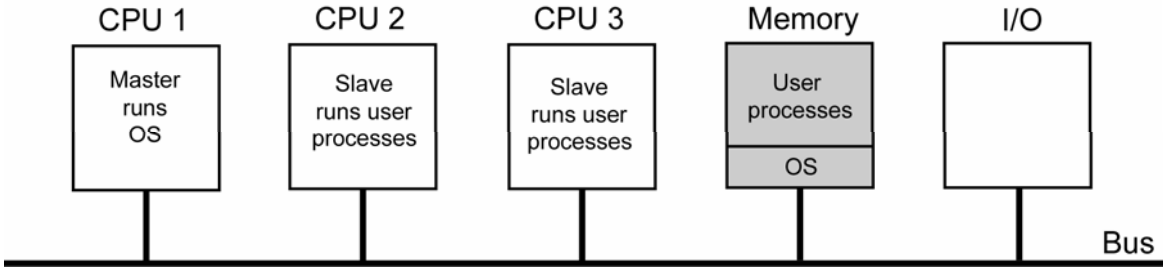


Multiprocesorové OS

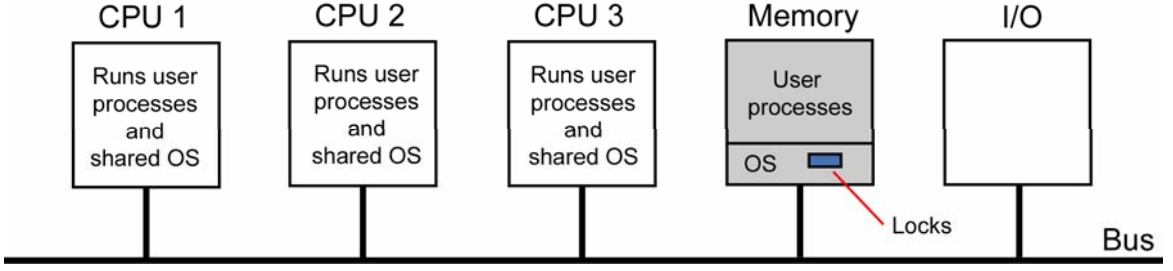
- **Každé CPU se svým vlastním OS** (nepožívá se)



- **Master-slave model (asymetrický)**

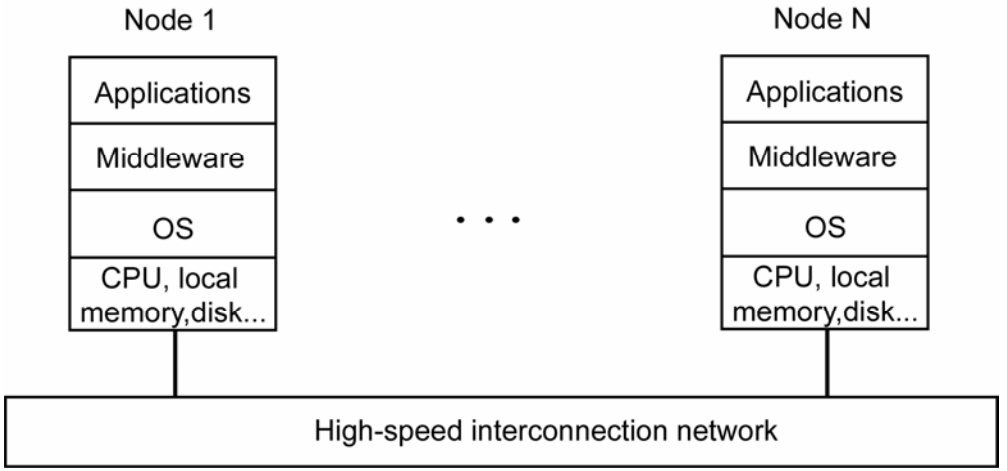


- **SMP (Symetric MultiProcessors)** – podpora v moderních OS



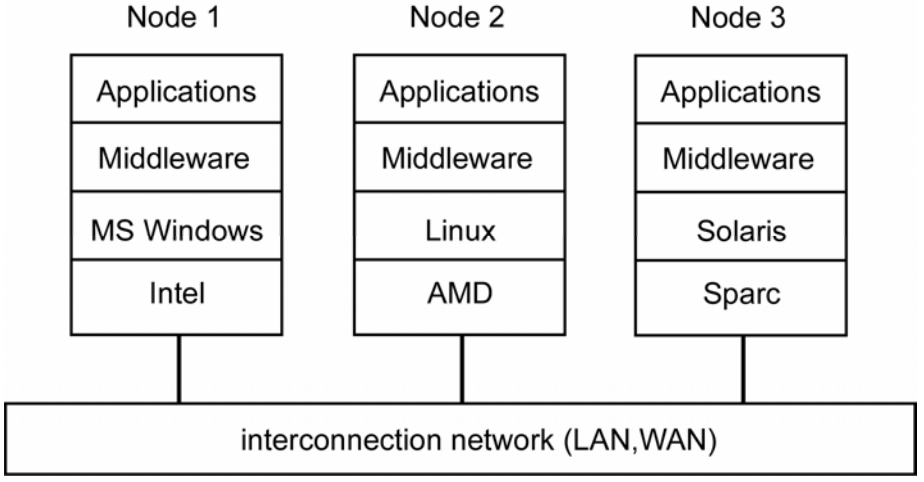
Multipočítače se zasíláním zpráv (cluster)

- **Základní uzel**
  - CPU, lokální paměť, síťové rozhraní, disk,…
  - ostatní periferie můžou chybět
- **Vysokorychlostní propojovací síť**
- **Všechny uzly**
  - používají stejný OS
  - sdílí společný systém souborů
  - jsou společně administrovány
- **Komunikace mezi CPU** pomocí zasílání zpráv (10-50 μs)



Distribuované systémy

- **Základní uzel**
  - CPU, paměť, disk, síťové rozhraní
  - ostatní periferie (obrazovka, klávesnice, ….)
- **Běžná propojovací síť (LAN, WAN)**
- **Jednotlivé uzly**
  - požívají různé OS
  - mají různé systémy souborů
  - jsou administrovány individuálně
- **Komunikace mezi CPU** pomocí zasílání zpráv (10-50 ms)



Vlastnosti OS

- **Víceúlohový (multitasking, time-sharing)**
  - běh více úloh (procesů) ve sdílení času
  - ochrana paměti, plánování procesů
- **Vícevláknový (multithreading)**
  - proces se může skládat z několika současně běžících úloh (vláken)
  - přechod od plánování procesů na plánování vláken (thread)
- **Víceuživatelský (multi-user)**
  - možnost současné práce více uživatelů
  - identifikace a vzájemná ochrana uživatelů
- **Podpora multiprocesorových systémů (SMP)**
  - použití vláken v jádře a jejich plánování na různých CPU
- **Unifikované prostředí**
  - přenositelnost mezi platformami (90% jádra v jazyce C)

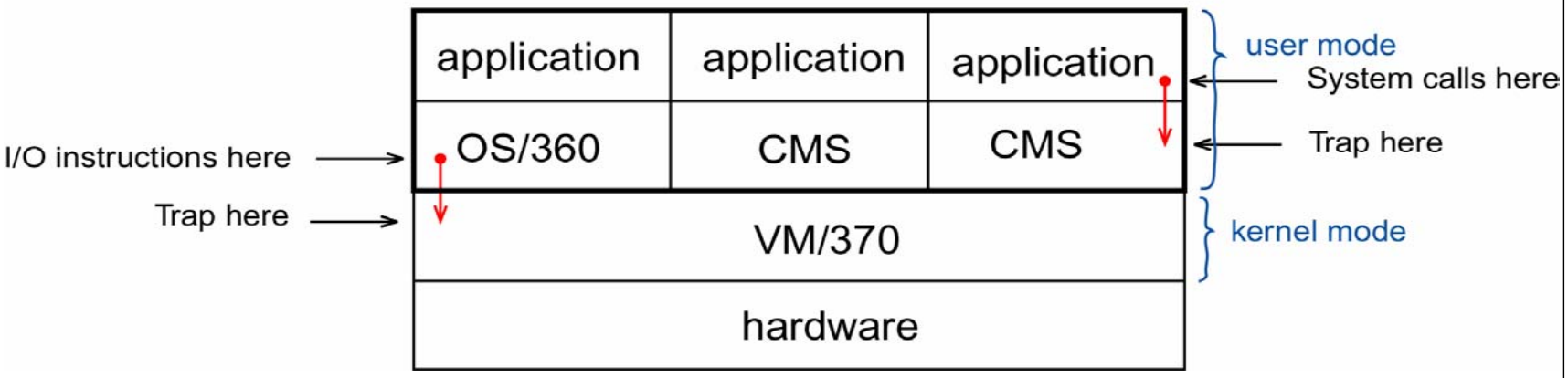
Struktura OS

- **Monolitický systém**
  - implementováno jako jeden proces
  - program je rozčleněn pouze na funkce/procedury
  - většinou vše běží v kernel modu

- **Vrstevný (layered) systém**
  - je rozdělen do hierarchických vrstev
  - každá vrstva řeší specifický úkol
  - přesně definované rozhraní mezi vrstvami
  - vnitřní vrstvy mohou být privilegovanější

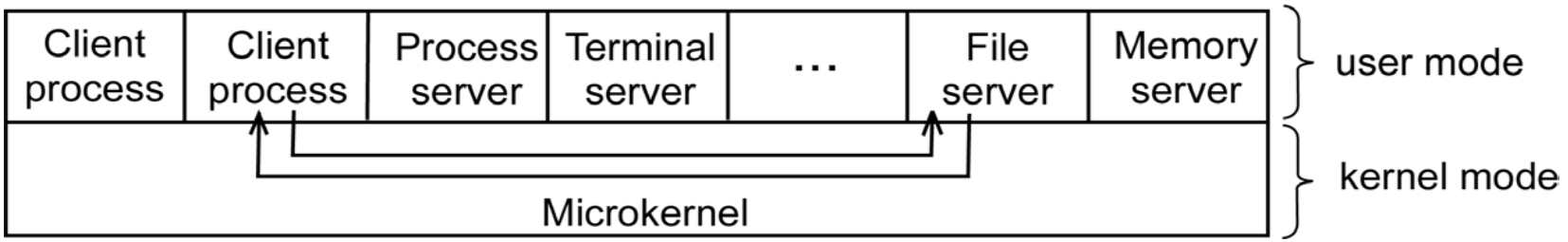
Struktura OS (2)

- **Virtuální stroj**
  - emuluje HW (přesná SW kopie HW)

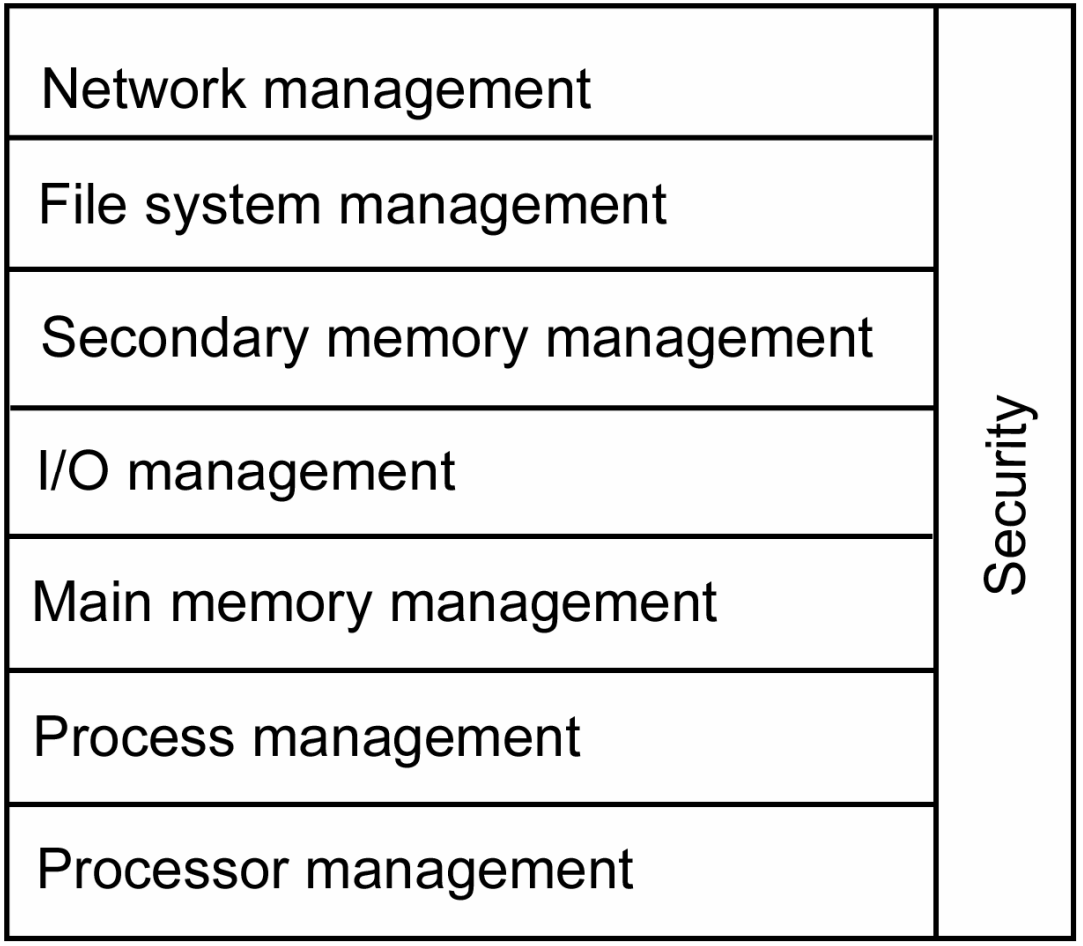


Struktura OS (3)

- **Mikrokernel**
  - v jádru je implementováno minimum služeb
  - poskytuje: přepínání kontextu, posílání správ,…
  - technologie klient/server
  - problém se systémovými servery používajících privilegované instrukce



Základní komponenty OS



Historie - Unix

konec 60. let	AT&T vyvíjí MULTICS
1969	AT&T Bell Labs - začátek OS Unix
začátek 70.let	AT&T vývoj OS Unix
1973	Unix implementován v jazyce C
kolem 1975	University of California at Berkley - UNIX (BSD)
začátek 80. let	Komerční zájem o OS Unix, DARPA AIX, HP-UNIX, Solaris
konec 80.let	Návrh standardů (POSIX, XPG, SVID), SVR4 UNIX
1991	Linus B. Torvalds vytváří jádro OS Linux

Historie - MS Windows

1980	8-bitový procesor Intel 8080 a Z80 **CP/M** (příkazová řádka)
1981	IBM PC - 16-bitový procesor Intel 8088 **MS-DOS 1.0** (jednoúlohový, jednouživatelský, příkazová řádka, systém souborů FAT)
1983	**MS-DOS 2.0** (vylepšená příkazová řádka, inspirace z Unixu)
1990	IBM PC/AT - Intel 386 **Windows 3.0, 3.1, 3.11** (grafické rozraní nad MS-DOS, inspirace z Apple Lisa)
1995	**Windows 95** (grafické rozraní, "32-bitový OS", virtuální paměť, správa procesů, víceúlohový, jednouživatelský, systém souborů FAT)
1998	**Windows 98** (vylepšení grafického prostředí a podpora Internetu)
2000	**Windows ME** (Windows Millennium Edition)

Historie - MS Windows (2)

1993	**Windows NT 3.1** (Windows New Technology) skutečně 32-bitový OS, víceúlohový OS, podpora vláken, virtuální paměť přenositelný (80x86, Pentium, Alpha, MIPS, PowerPC,...) systém souborů FAT32 nebo NTFS grafické rozhraní podobné Windows 95/98
1996	**Windows NT 4.0**
1999	**Windows 2000** (Windows NT 5.0) podpora SMP (symmetric multiprocessing), podpora až 32 CPU, až 64GB fyzické paměti
2001	**Windows XP** (Windows NT 5.1) 32-bitový/64-bitový OS, podpora až 2CPU a 4GB/16GB
2003	**Windows Server 2003** (Windows NT 5.2) 32-bitový/64-bitový OS, podpora až 32/64CPU a 64GB/1024GB (Itanium)
2006	**Windows Vista**

MSDN Academic Alliance

- Stahování a používání SW od Microsoftu pro nekomerční účely
- Zaměstnanci a studenti KP mohou stahovat zadarmo
- SW lze používat i po skončení studia pro nekomerční účely
- **Registrace**
  - https://msdn62.e-academy.com/msdnaa_fx9297/index.cfm?loc=login/register
- **Odpovědná osoba:**
  - Ing. Miloš Puchta, CSc.
  - email: puchta@fel.cvut.cz

Operační systémy

Přednáška 2: Procesy a vlákna

Procesy

- Všechen běžící software v systému je organizován jako množina běžících procesů.

- **(Sekvenční) proces**
  - Abstrakce běžícího programu.
  - Sekvence výpočetních kroků **závisí** pouze na **výsledcích předchozích kroků** a na **vstupních datech**.

- **Paralelní sekvenční procesy** (paralelní program)
  - Množina sekvenčních procesů běžících „současně“.
  - Procesy mohou běžet na jednoprocesorovém systému (**pseudo parallelism**) nebo na multiprocesorovém systému (**real parallelism**).

- **Výsledek** paralelního deterministického programu by **neměl záviset na rychlosti provádění** jednotlivých procesů.

Příklad – hledání maximálního prvku

- A – pole *n* čísel (*n* ~ 10 000)
- B – pomocné pole *p* čísel (*p* ~ 10)
- Funkce max(*A*, i, j) vrátí maximální prvek z množiny A[*i*] …A[*j*]
- Funkce pmax() je podobná jako max(), ale spouští se v novém procesu.
- **Sekvenční řešení**:

```
main()
{  …  M = max (A, 0, n-1)  …  }
```

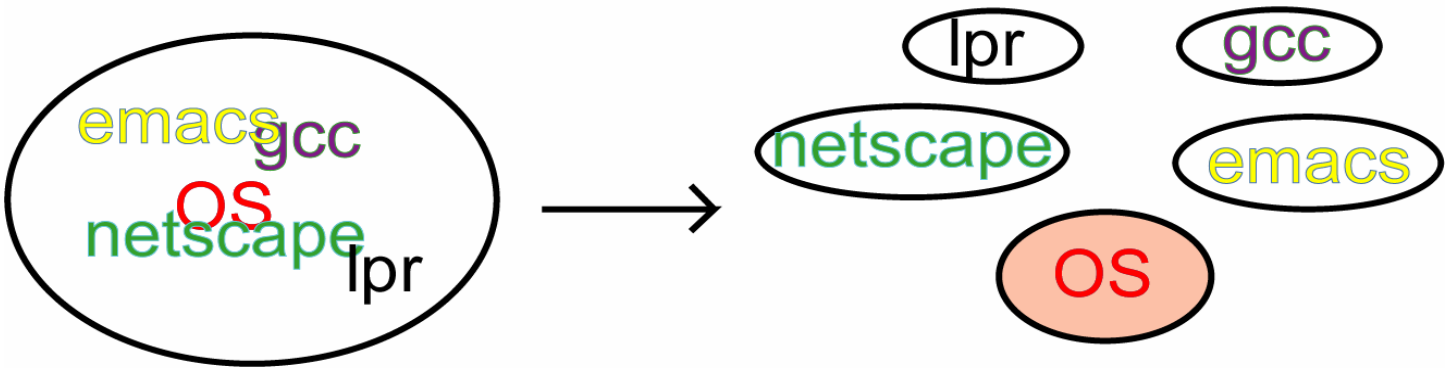
- **Paralelní řešení**:

```
main()
{ …
    for ( i = 0; i < p;  i++) { B[i] = pmax (A, i*n/p, (i+1)*n/p-1) };
    max (B, 0, p-1) ;
…}
```

- Bude výsledek v obou případech stejný?

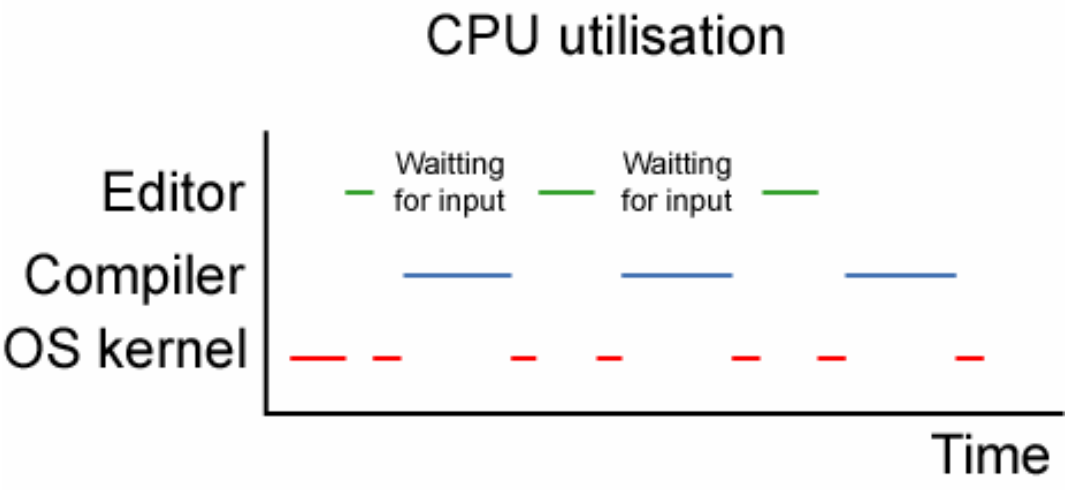
Proč procesy - jednoduchost

- **Systém provádí spoustu různých věcí**



- **Jak to zjednodušit?**
  - Z každé jednotlivé věci udělat izolovaný proces.
  - OS se zabývá v jednom okamžiku pouze jednou věcí.
  - Univerzální trik pro správu složitých problémů: **dekompozice problému**.

Proč procesy - rychlost

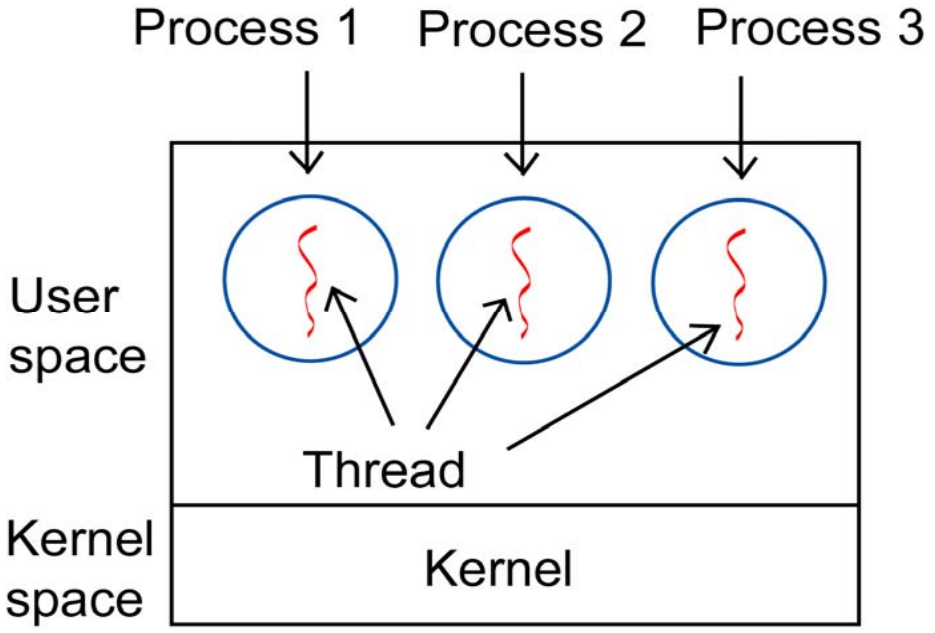


- **V/V paralelismus**
  - Zatímco jeden proces čeká na dokončení V/V operace jiný proces může používat CPU.
  - Překrývání zpracování: dělá z 1 CPU více CPU.
  - Reálný paralelismus.

Procesový model

- Každý proces **alokuje příslušné prostředky** (adresový prostor obsahující kód a data, otevřené soubory, reakce na signály, ...), má **identitu** (PID, PPID, vlastníka, seznam potomku,…)…

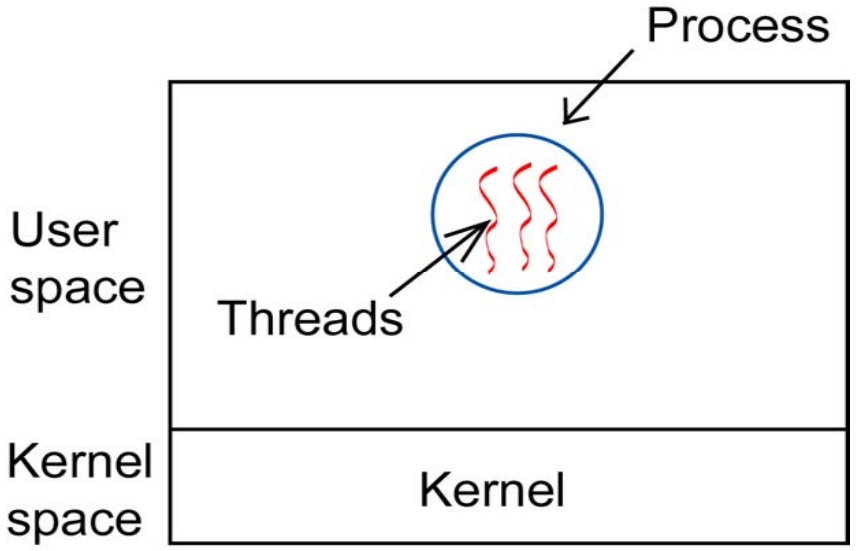
- Implicitně obsahuje **jedno vlákno výpočtu**.



Vláknový model

- Oddělení alokace prostředků a samotný výpočet.
- **Proces** slouží k **alokaci společných prostředků**.
- **Vlákna** jsou **jednotky plánované pro spuštění** na CPU.

- **Vlákno** má svůj vlastní **program counter** (pro uchování informace o výpočtu), **registry** (pro uchování aktuálních hodnot), **zásobník** (který obsahuje historii výpočtu), **lokální proměnné**, ale **ostatní prostředky** a **identita jsou sdílené**.



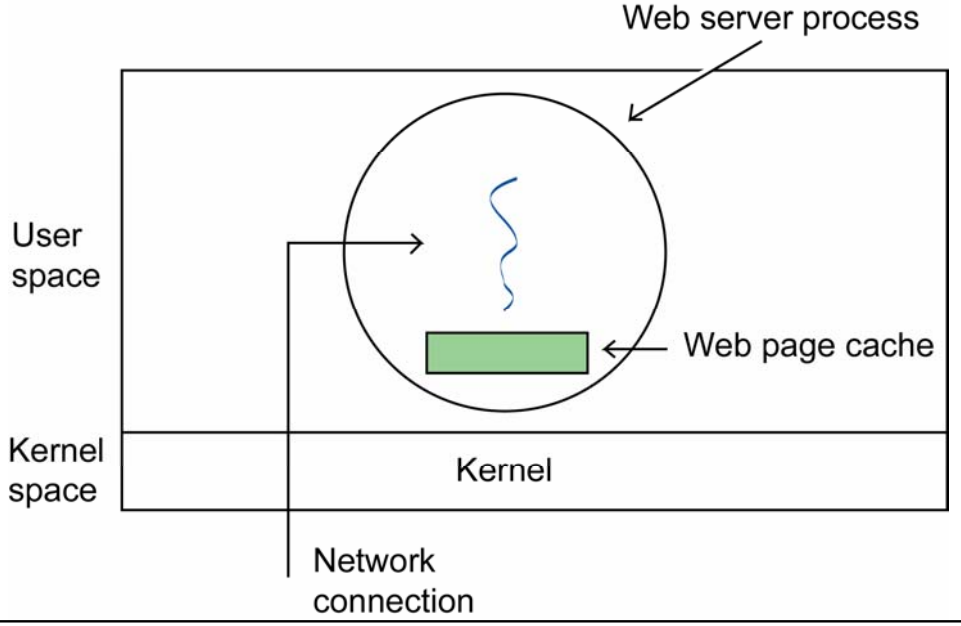
Vláknový model (2)

- Jednotlivá vlákna v daném procesu **nejsou nezávislá** tak jako jednotlivé procesy.

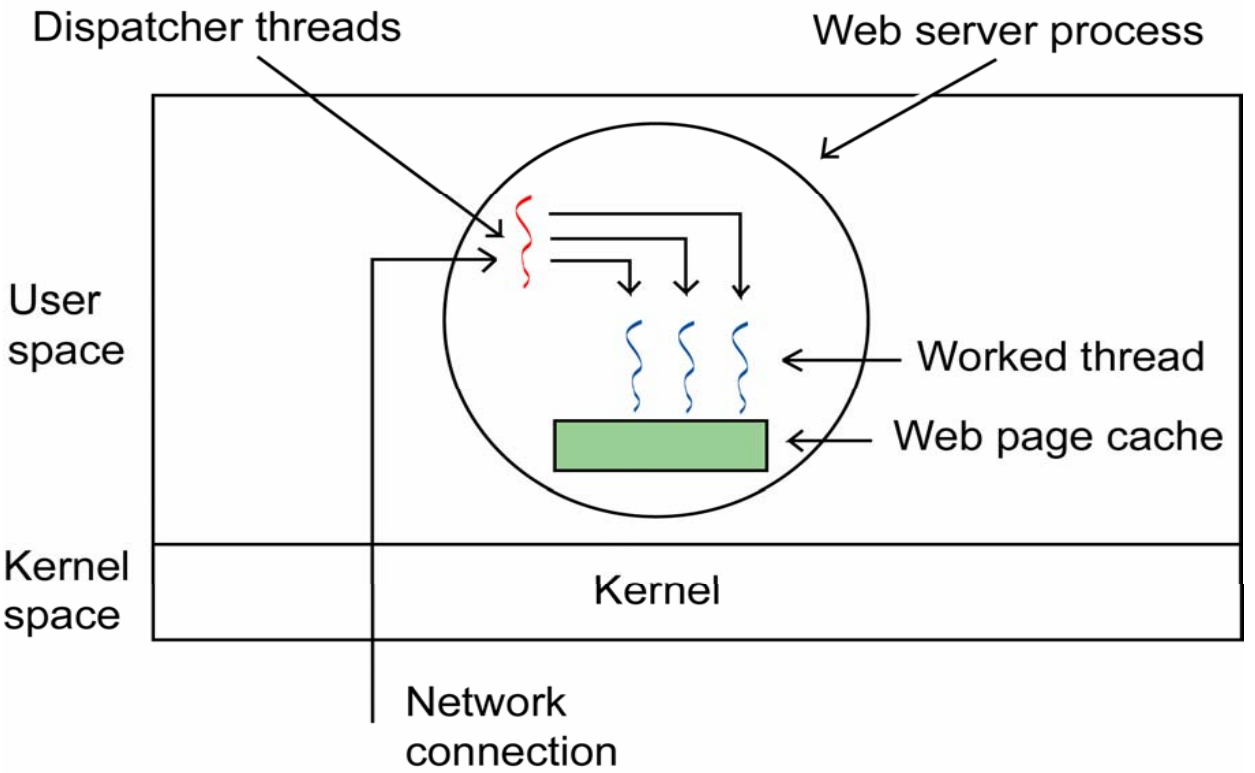
- Všechny vlákna v procesu **sdílí** stejný adresový prostor, stejné otevřené soubory, potomky, reakce na signály, …

Příklad: jednovláknový Web Server

- **Klient**
  - pošle požadavek na konkrétní web. stránku
- **Server**
  - ověří zda klient může přistupovat k dané stránce
  - načte stránku a pošle obsah stránky klientovi
- **Často používané stránky** zůstávají uloženy v **hlavní paměti**, abychom minimalizovali čtení z disku.



Příklad: vícevláknový Web Server



Příklad: vícevláknový Web Server (2)

- **Dispatcher thread:** čte příchozí požadavky, zkoumá požadavek, vybere nevyužité pracovní vlákno a předá mu tento požadavek.

- **Worked thread:** načte požadovanou stránku z hlavní paměti nebo disku a pošle ji klirntovi.

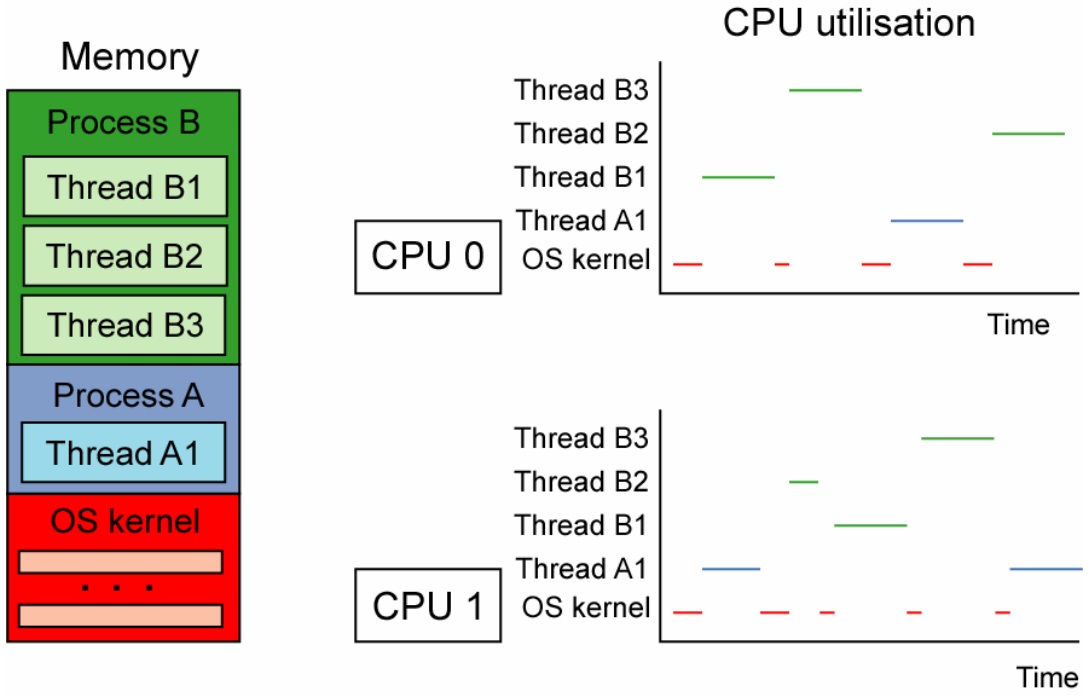
Dispatcher thread

```
while (TRUE) {
  get_next_request(&buf);
  handoff_work(&buf);
}
```

Worked thread

```
while(TRUE) {
  wait_for_work(&buf);
  look_for_page_in_cache(&buf,&page);
  if (page_not_in_cache(&page))
    read_page_from_disk(&buf,&page);
  return_page(&page);
}
```

Přepínání kontextu



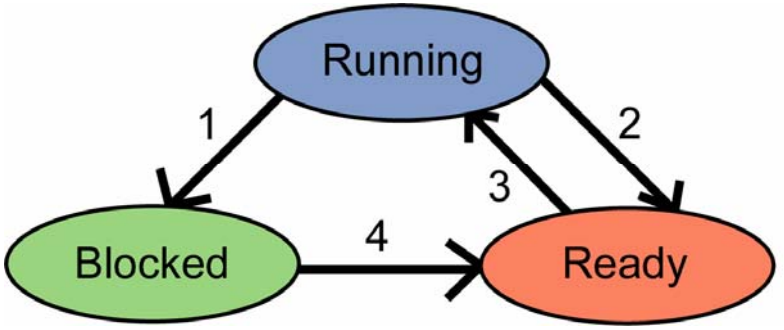
- **Pseudo paralelismus:** vlákna běží (pseudo) paralelně na jednoprocesorovém systému díky **přepínání kontextu**, procesor střídavě provádí kód jednotlivých vláken (**multiprogramming, timesharing, multiprocessing**).
- **Skutečný hardwarový paralelismus:** každé vlákno běží na svém procesoru (multiprocesorový systém).

Příklad: přepínání kontextu

- Hodnota časového kvanta je kompromisem mezi **potřebami uživatele** (malý čas odezvy) a **potřebami systému** (efektivní přepínání kontextu), přepínání kontextu trvá zhruba 2-5 ms).

- Časové kvantum v OS je zhruba 10-100 ms (např. parametr jádra **time slice**).

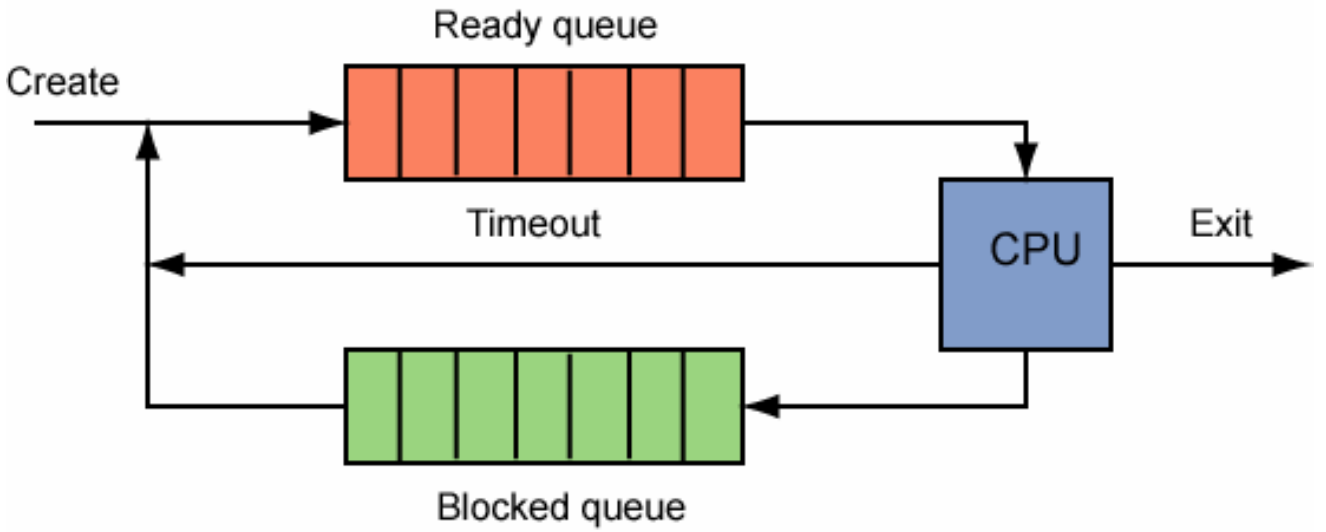
Stavy vlákna – třístavový model



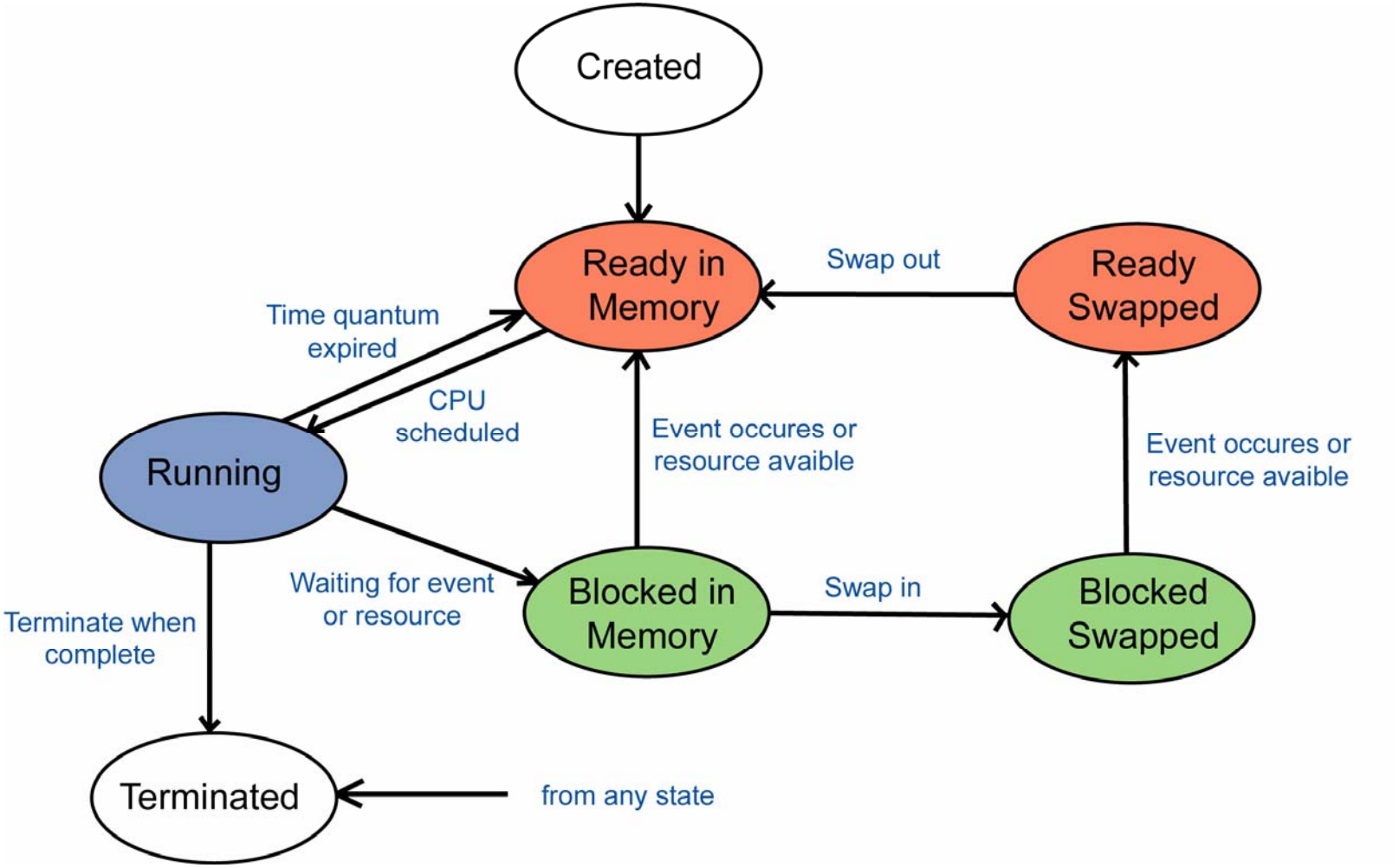
1. Thread blocks for input
2. Scheduler picks another thread
3. Scheduler picks this thread
4. Input becomes available

- **Základní stavy vlákna**
  - **Running**: v tomto okamžiku právě používá CPU.
  - **Ready**: připraveno použít CPU, dočasně je vlákno zastaveno a čeká až mu bude přiřazeno CPU.
  - **Blocked**: neschopné použít CPU v tomto okamžiku, čeká na nějakou externí událost (např. načtení dat z disku,…).

Implementace pomocí front



Stavy vlákna – pětistavový model



Vytvoření procesu

- Nový proces se vytvoří když exitující proces zavolá příslušné **systémové volání** (např. **`fork()`** and **`exec()`** v Unix, nebo **`CreateProcess()`** v MS Windows).

- **Vytvoření**
  - OS inicializuje v jádře datové struktury spojené s novým procesem.
  - OS nahraje kód a data programu z disku do paměti a vytvoří prázdný systémový zásobník pro daný proces.

- **Klonování**
  - OS zastaví aktuální proces a uloží jeho stav.
  - OS inicializuje v jádře datové struktury spojené s novým procesem.
  - OS udělá kopii aktuálního kódu, dat, zásobníku, stavu procesu,...

Příklad: vytvoření procesu v Unixu

```
        . . .
Pid = fork() ;

if(pid != 0) {
        /* parent */
        wait(pid) ; /* wait for child to finish */
else {
        /* child process */
        exec("ls") ;  /* exec does not return */
}
        . . .
```

Příklad: vytvoření procesu v Unixu (2)

- Nový proces vzniká použitím systémových volání:

- **`fork()`**
  - vytvoří nový proces, který je kopií procesu, z kterého byla tato funkce zavolána
  - v rodičovském procesu vrací funkce PID potomka (v případě chyby -1)
  - v potomkovi vrací funkce 0
  - nový proces má jiné PID a PPID, ostatní vlastnosti dědí (např. EUID, EGID, prostředí, ...) nebo sdílí s rodičem (např. soubory, ...)
  - kódový segment sdílí potomek s rodičem
  - datový a zásobníkový segment vznikají kopií dat a zásobníku rodiče

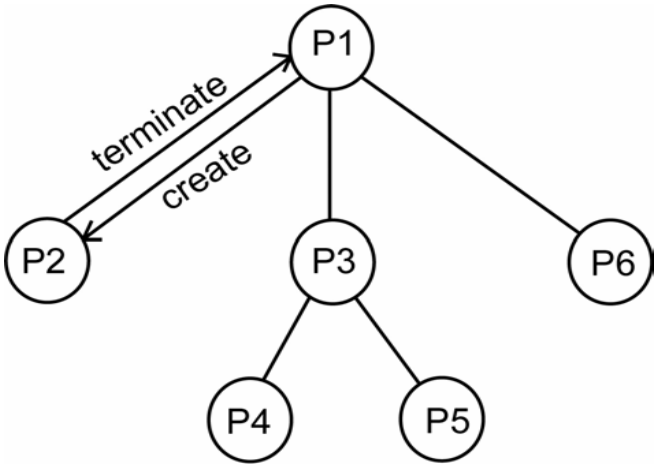
Příklad: vytvoření procesu v Unixu (3)

- **`exec()`**
  - v procesu, ze kterého je funkce volána, spustí nový program (obsah původního procesu je přepsán novým programem)
  - atributy procesu se nemění (PID, PPID, ...)

Ukončení procesu

- **Normal exit** (dobrovolné)
  - Když proces dokončí svou práci, použije systémové volání, aby řekl OS, že končí (např. **`exit()`** v Unixu nebo **`ExitProcess()`** v MS Windows).
- **Error exit** (dobrovolné)
  - Například když proces zjistí fatální chybu (např. žádný vstupní soubor,...).
- **Fatal error** (nedobrovolné)
  - Chyba způsobená procesem, často např. díky chybě v programu. OS proces násilně ukončí.
- **Ukončení jiným procesem** (nedobrovolné)
  - Proces použije systémové volání, aby řekl OS o ukončení nějakého jiného procesu (např. **`kill()`** v Unixu nebo **`TerminateProcess()`** v MS Windows).

Hierarchie procesů



- V některých systémech, když proces vytvoří další proces, rodičovský proces a potomek jsou jistým způsobem svázány (např. Unix: **vztah parent process – children process**).
- Potomek může **zdědit** některé rysy od svého rodiče (např. kód procesu, globální data,…).
- Na druhé straně, každý nový proces má své vlastní struktury (**vlákna→ svůj vlastní zásobník, reakce na signály, lokální data**).

Příklad: hierarchie v Unixu

ptree -a $$

1 /sbin/init

401 /usr/lib/ssh/sshd

14905 /usr/lib/ssh/sshd

14912 /usr/lib/ssh/sshd

14914 -bash

10759 ptree -a 14914

Vytvoření a ukončení vlákna

- Procesy se spouští s jedním vláknem.
- Toto vlákno může **vytvářet další vlákna** pomocí knihovní funkce (např. *pthread_create(name_of_function)*).
- Když chce vlákno skončit, může se opět **ukončit** pomocí knihovní funkce (např. *pthread_exit()*).

Příklad: POSIX vlákna

```
...
void *kod_vlakna(void *threadid)
{ printf("ID vlakna: %d\n", threadid);
  sleep(60);
  pthread_exit(NULL);
}

int main()
{ pthread_t threads[NUM_THREADS];
  int rc, i;
  for(i=0; i<NUM_THREADS; i++){
    rc = pthread_create(&threads[i], NULL, kod_vlakna, (void *) i);
    if (rc != 0){ perror("Chyba ve funkci pthread_create()"); exit(1); }
  }
  ...
}
```

Systémové řídící struktury

- **Tabulky paměti** (memory table)
  - informace o fyzické a virtuální paměti
- **Tabulky V/V** (I/O table)
  - informace V/V zařízeních (alokace, stav,…)
- **Tabulky souborů** (file table)
  - informace o otevřených souborech
- **Tabulky procesů** (process table)
  - informace o existujících procesech

Implementace procesu

- OS zpravuje tabulku (pole struktur), která se nazývá **tabulka procesů**, s jednou položkou pro jeden proces, nazývanou **process control block (PCB)**.

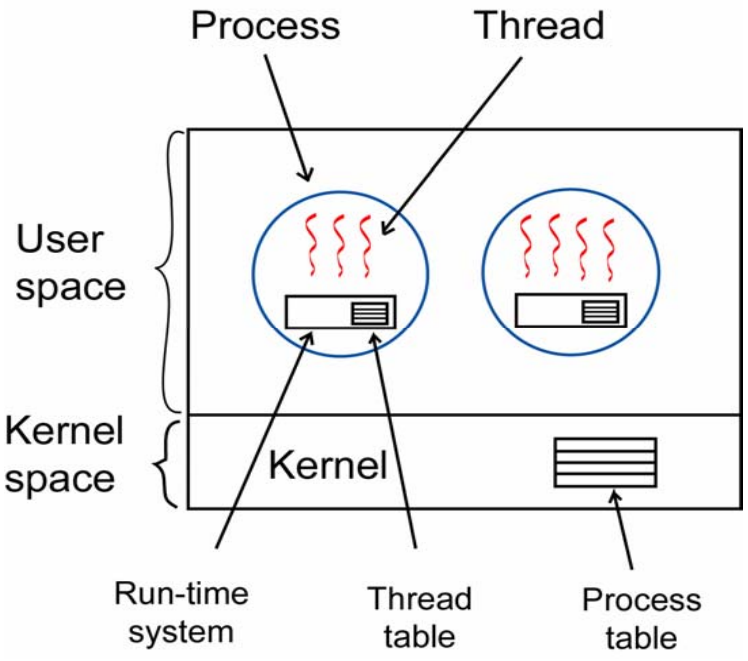
- **PCB** obsahuje **informace o procesu**, které jsou nutné pro správu procesů.

- Například:
  - V Unixu, maximální velikost tabulky procesů je definována parametrem jádra **`nproc`**.
  - PCB má v Unix přibližně 35 položek.

Položky PCB

- **Informace pro identifikaci procesu** (process identification)
  - jméno procesu, číslo procesu (PID), rodičovský proces (PPID), vlastník procesu (UID, EUID), seznam vláken, …
- **Stavové informace procesoru** (processor state information) – pro každé kernel vlákno
  - hodnoty viditelných registrů CPU,
  - hodnoty řídících a stavových registrů CPU (program counter, program status word (PSW), status information, …)
  - ukazatelé na zásobníky, …
- **Informace pro správu procesu** (process control information)
  - Vlákna:
    - stav vlákna, priorita, informace nutné pro plánování
    - informace o událostech, na které proces čeká,
    - informace pro komunikaci mezi procesy, …
  - Proces:
    - informace pro správu paměti (ukazatel na tabulku stránek,…)
    - alokované a používané prostředky,…

Implementace vláken v uživatelském prostoru



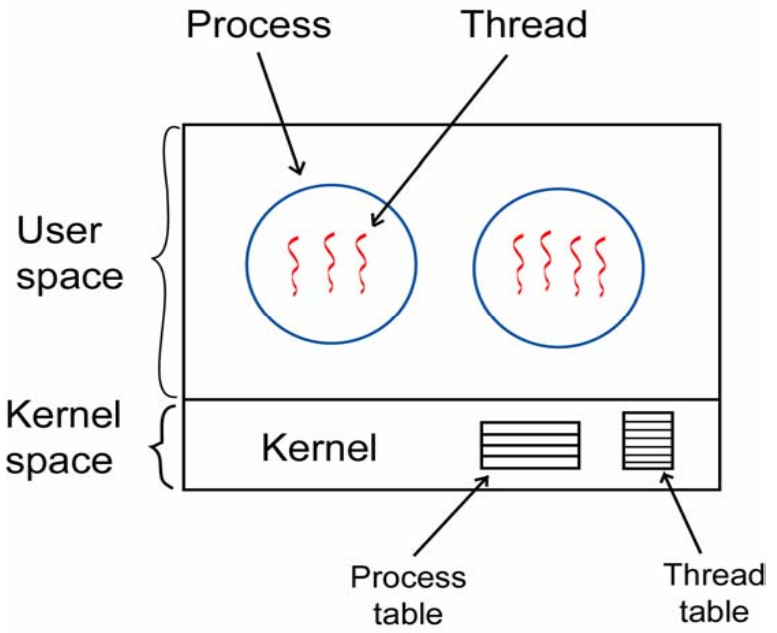
- **Run-time system**: množina funkcí, která spravuje vlákna.
- Vlákna jsou implementována pouze v uživatelském prostoru.
- Jádro o vláknech nemá žádné informace.

Implementace vláken v uživatelském prostoru (2)

- **Výhody**
  - Vlákna mohou být implementována v OS, které nepodporuje vlákna.
  - Rychlé plánování vláken.
  - Každý proces může mít svůj vlastní plánovací algoritmus.

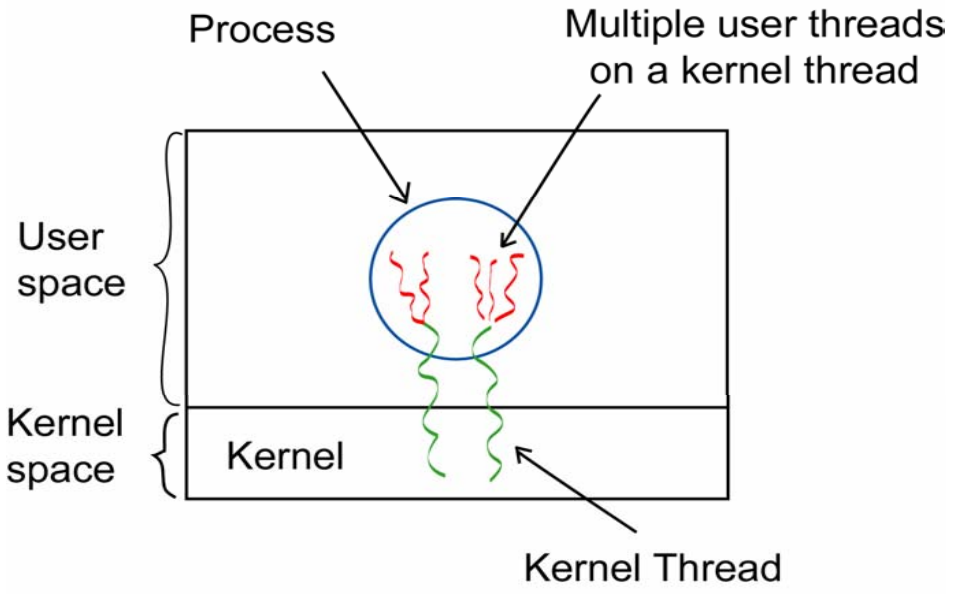
- **Nevýhody**
  - Jak budou implementována blokující systémová volání?
    (změna systémových volání na neblokující nebo požití systémového volání **`select`**)
  - Co se stane když dojde k výpadku stránky?
  - Žádný clock interrupt uvnitř procesu.
    (jedno vlákno může okupovat CPU během celého časového kvanta procesu)

Implementace vláken v prostoru jádra



- Jádro má tabulku vláken, která obsahuje informace o všech vláknech v systému.
- **Výhody**
  - Žádný problém s blokujícími systémovými voláními.
- **Nevýhody**
  - Vytváření, ukončování a plánování vláken je pomalejší.

Hybridní implementace vláken



- Jádro se stará pouze o **kernel-level threads** a plánuje je.
- Některé kernel-level threads mohou mít user-level threads.
- **User-level threads** jsou vytvářená, ukončovaná a plánovaná uvnitř procesu.
- Např. Solaris, Linux, MS Windows

Hybridní implementace vláken (2)

- Anderson et. al. 1992.
- **Kombinuje** výhody **uživatelských vláken** (dobrá výkonnost) s výhodami **kernel vláken** (jednoduchá implementace).

- **Princip**:
  - Jádro přidělí určitý počet **virtuálních procesorů** každému procesu.
  - (Uživatelský) **run-time system** může **alokovat** vlákna na virtuální procesor, **požádat o další** nebo **vrátit** virtuální procesory jádru.

Hybridní implementace vláken (3)

- **Problem**: blokující systémová volání.
- **Řešení**:
  - Když jádro ví, že **bude vlákno zablokováno**, jádro aktivuje **run-time system a dá mu o tom vědět** (tzv. „upcall“).
  - Aktivovaný run-time systém **může naplánovat další** ze svých vláknen.
  - Když původní vlákno je opět ve stavu „ready“, jádro provede znova upcall, aby to oznámilo run-time systému.

Solaris

- **Process**: normální Unixový proces.
- **User-level threads** (ULT): implementovaný pomocí knihovny vláken v adresovém prostoru procesu (neviditelný pro OS).
- **Lightweight processes** (LWP): mapování mezi ULT a kernel vlákny. Každý LWP podporuje jedno nebo více ULT a je mapováno na jedno kernel vlákno.
- **Kernel threads**: základní jednotky, které mohou být plánovány a spuštěny na jednom nebo více CPU.

Vlákna ve Windows XP

- **Job**: množina procesů, které sdílejí kvóty a limity (maximální počet procesů, čas CPU, paměť, …).
- **Process**: jednotka, která alokuje zdroje. Každý proces má aspoň jedno vlákno (thread).
- **Thread**: jednotka plánována jádrem.
- **Fiber**: vlákno spravované celé v uživatelském prostoru.

Otázky k zamyšlení

- Jedno z vláken procesu vytvoří nový proces pomocí funkce fork(). Kolik vláken bude v nově vzniklém procesu?
- Jak dlouho bude trvat přepnutí kontextu mezi dvěma vlákny téhož procesu a mezi dvěma vlákny různých procesů? Proč?
- Uživatelské vlákno se pokusí číst data ze souboru. Co se stane s ostatními vlákny procesu?

Operační systémy

Přednáška 3: Komunikace mezi procesy

Časově závislé chyby

- Dva nebo několik procesů/vláken používá (čte/zapisuje) **společné sdílené prostředky** (např. sdílená paměť, sdílení proměnné, sdílené soubory,…).
- **Výsledek** výpočtu **je závislý na přepínání kontextu** jednotlivých procesů/vláken, které používají sdílené prostředky.
- Velmi špatně se detekují (náhodný výskyt)!

Příklad: časově závislé chyby

- Program v C přeložený do asembleru 16-bitového procesoru.

32-bit shared variable

```
long var = 0x0000ffff;          # 32 bit shared variable !
                                # two words !
```

Thread A

```
…
var++;
…
```

```
ADD var, #1                     # ffff + 1 = carry bit
…       # if context switch comes here the value of var is 0
ADDC var+2, #0                  # add carry bit to 0000
…       # if context switch comes here the value of var != 0
```

Thread B

```
if (var == 0) { ...}
else { … }
```

```
MOV R0, var         # the result of the test depends
OR    R0, var+2     # on the context switch time
```

Příklad: časově závislé chyby (2)

- Dva uživatelé editují stejný soubor (např. v Unixu pomocí editoru vi).

User 1: vi f.txt

User 2: vi f.txt

Kritické sekce

- **Kritická sekce**
  - Část programu, kde procesy používají sdílené prostředky (např. sdílená paměť, sdílená proměnná, sdílený soubor, …).
- **Sdružené kritické sekce**
  - Kritické sekce dvou (nebo více) procesů, které se týkají stejného sdíleného prostředku.
- **Vzájemné vyloučení**
  - Procesům není dovoleno sdílet stejný prostředek ve stejném čase.
  - Procesy se nesmí nacházet ve sdružených sekcích současně.

Korektní paralelní program

- **Nutné podmínky**
  1. Dva procesy se nesmí nacházet současně ve stejné sdružené sekci.
  2. Žádné předpoklady nesmí být kladeny na rychlost a počet procesorů.
  3. Pokud proces běžící mimo kritickou sekci nesmí být blokován ostatní procesy.
  4. Žádný proces nesmí do nekonečna čekat na vstup do kritické sekce.

Bernsteinovy podmínky

$$\mathrm{R}(p) \cap \mathrm{W}(q) = \varnothing,$$
$$\mathrm{W}(p) \cap \mathrm{R}(q) = \varnothing,$$
$$\mathrm{W}(p) \cap \mathrm{W}(q) = \varnothing,$$

R(i) - všechny sdílené proměnné pro čtení použité v procesu i,
W(i) - všechny sdílené proměnné pro zápis použité v procesu i.

- **Sdílené proměnné** mohou být **pouze čteny.**
- Příliš přísné => pouze teoretický význam.

Zákaz přerušení (DI)

- CPU je přidělováno postupně jednotlivým procesům za pomoci přerušení od časovače nebo jiného přerušení.
- Proces **zakáže všechna přerušení před vstupem** do kritické sekce a opět je **povolí až po opuštění** kritické sekce.

- **Nevýhoda**:
  - DI od jednoho uživatele blokuje i ostatní uživatele.
  - Ve víceprocesorovém systému, DI má efekt pouze na aktuálním CPU.
  - Zpomalí reakce na přerušení.
  - Problém se špatně napsanými programy (zablokují CPU).

- Užitečná technika uvnitř jádra OS (ale pouze na krátký čas).
- Není vhodná pro běžné uživatelské procesy!!!

Aktivního čekání vs. blokování

- Pouze **jeden proces může vstoupit** do kritické sekce.
- **Ostatní procesy musí počkat** dokud se kritická sekce neuvolní.
- **Aktivní čekání**
  - sdílená proměnná indikuje obsazenost kritické sekce
  - proces ve smyčce testuje aktuální hodnotu proměnné do okamžiku než se sekce uvolní
- **Blokování**
  - proces provede systémové volání, které ho zablokuje do okamžiku než se sekce uvolní

Lock proměnná

- Vzájemné vyloučení pomocí **(sdílené) lock proměnné**, kterou proces nastaví když vstupuje do kritické sekce.

```
int lock=0;
. . .
while (lock == 1);          /* busy waiting is doing */
                            /* critical place for context switching */
lock=1;
critical_section();
lock=0;
. . .
```

- **Problém**
  - pokud dojde k přepnutí kontextu v označeném místě, může vstoupit do kritické sekce další proces => **špatné řešení**
  - testování a nastavení proměnné **není atomické**

Striktní střídání

```
/* Process A */
. . .
while (TRUE) {
  while (turn != 0);   /* wait */
  critical_section();
  turn=1;
  noncritical_section();
}
```

```
/* Process B */
. . .
while (TRUE) {
  while (turn != 1);   /* wait */
  critical_section();
  turn=0;
  noncritical_section();
}
```

- **Nevýhody**
  - Jeden proces může zpomalit ostatní procesy.
  - Proces nemůže vstoupit do kritické sekce opakovaně (je porušen 3.bod z nutných podmínek ...).

Petersonův algoritmus

- T. Dekker (1968) navrhl algoritmus, který nevyžadoval striktní střídání.
- G. L. Peterson (1981) popsal jednodušší verzi tohoto algoritmu.
- L. Hoffman (1990) zobecnil řešení pro *n* procesů.

Petersonův algoritmus (2)

```
#define FALSE 0
#define TRUE  1
#define N     2                          /* number of processes */

int turn;                                /* whose turn is it? */
int interested[N];                       /* all values initially FALSE */

void enter_section (int process)         /* process is 0 or 1 */
{
        int other;                       /* number of other processes */

        other = 1 - process;             /* the opposite of process */
        interested[process] = TRUE;      /* show that you are interested */
        turn = process;                                  /* set flag */
        while (turn == process && interested[other]);    /* busy waiting */
}

void leave_section (int process)         /* process: who is leaving */
{
        interested[process] = FALSE;     /* indicate departure from CS */
}
```

Instrukce TSL

- **Test and Set Lock** (TSL) instrukce **načte** obsah slova z dané adresy v paměti do registru a **nastaví** obsah slova na nenulovou hodnotu.

- CPU provádějící TSL instrukci **zamkne paměťovou sběrnici,** aby znemožnilo ostatním CPU v přístupu do sdílené paměti dokud se instrukce TSL nedokončí.

- TSL je atomická instrukce => **korektní hardwarové řešení**.

- **Výhody**
  - TSL může být použita při synchronizaci v multiprocesorových systémech se sdílenou pamětí.

Instrukce TSL (2)

```
enter_section:
   TSL REGISTER, LOCK      | copy LOCK to REGISTER and set LOCK to 1
   CMP REGISTER,#0         | was LOCK zero?
   JNE enter_section       | if it was non zero, LOCK was set, so loop
   RET                     | return to caller, critical section entered


leave_section:
         MOVE LOCK,#0      | store a 0 in LOCK
         RET               | return to caller
```

Nevýhody aktivního čekání

- **Plýtvání časem procesoru**.

- Pokud OS používá prioritní plánování, potom může vzniknout **inverzní prioritní problém**
  - proces A má nižší prioritu a nachází se v kritické sekci
  - proces B má vyšší prioritu a čeká pomocí aktivního čekání na vstup do kritické sekce
  - pokud přiřazená priorita je fixní **dojde k uváznutí**

Vzájemné vyloučení pomocí blokování

- Lepší způsob než zákaz přerušení nebo aktivní čekání.
- Procesy, které nemohou vstoupit do kritické sekce jsou **zablokovány** (jejich stav se změnil na „blocked“ a jsou umístěny do čekací fronty).
- Blokující a deblokující operace jsou obvykle **implementovány jako služby jádra OS**.

Sleep a Wakeup

- **`Sleep()`**
  - Systémové volání, které zablokuje proces, který ho zavolal.
  - Zakáže alokování CPU pro tento proces a přesune ho do fronty kde bude čekat.

- **`Wakeup(proces)`**
  - Opačná operace, proces je uvolněn z fronty čekajících procesů a bude mu opět přidělováno CPU.

- **Problém producenta a konzumenta**
  - Jeden z klasických synchronizačních problémů, který demonstruje problémy paralelního programování.
  - **Producent** produkuje nějaká data a vkládá je do sdílené paměti.
  - **Konzument** vybírá data ze sdílené paměti.

Sleep a Wakeup (2)

```
define N 100                                    /* number of slots in the buffer */
int count = 0;                                  /* number of items in buffer */
```

```
void producer(void)
{     int item;
      while(TRUE){
            item = produce_item();
            if (count == N) sleep();                    /* if buffer is full, go to sleep */
            insert_item(item);
            count = count + 1;
            if (count == 1) wakeup(consumer);       /* was buffer empty? */ }
}
```

```
void consumer(void)
{     int item;
      while (TRUE){
            if (count == 0)                         /* critical point for context switching */
                  sleep();                          /* if buffer is empty, go to sleep */
            remove_item(&item)
            count = count – 1;
            if (count == N - 1) wakeup(producer);  /* was buffer full? */
            consume_item(&item);}
}
```

Sleep a Wakeup (3)

- Abychom se vyhnuli časově závislým chybám, musí být zaručeno, že nedojde k přepnutí kontextu v označeném místě.
- Tato podmínka není v tomto řešení garantována.
- **Problém**
  - Operace ***wakeup()*** je zavolána na proces, který není ještě uspán. Co se stane???
- **Řešení**
  - Wakeup waiting bit.
  - Když je ***wakeup()*** zavolána na proces, který ještě nespí, tento bit je nastaven.
  - Při pokusu uspat proces, který má nastaven tento bit, proces se neuspí, ale pouze se resetuje daný bit.

Operační systémy

Přednáška 4: Komunikace mezi procesy

Semafory

- Datový typ semafor obsahuje **čítač** a **frontu čekajících procesů**.

- Nabízí tři základní operace:
  - **`Init()`:** Čítač se nastaví na zadané číslo (většinou 1) a fronta se vyprázdní.
  - **`Down()`:** Pokud je čítač větší než nula, potom se sníží o jedničku. V opačném případě se volající proces zablokuje a uloží do fronty.
  - **`Up()`:** Pokud nějaké procesy čekají ve frontě, potom se první z nich probudí. V opačném případě se čítač zvětší o jedničku.

- Každá z těchto instrukcí je prováděna **atomicky** (tzn. že během jejího provádění nemůže být přerušena).

Semafory (2)

- Počáteční hodnota čítače semaforu určuje kolik procesů může sdílet nějaký prostředek současně.

- Z počáteční a aktuální hodnoty lze potom určit, kolik procesů je v daném okamžiku v kritické sekci.

- **Binární semafor**
  - Je často nazýván **mutex** (mutual exclusion).
  - Čítač semaforu je na začátku nastaven na jedničku.
  - Umožňuje jednoduše synchronizovat přístup do kritické sekce.

Kritická sekce hlídaná semaforem

```
typedef int semaphore;
semaphore mutex = 1;          /* control access to critical section(CS)*/
```

```
void process_A(void)
{
  while (TRUE) {
    ...
    down(&mutex);  /* enter CS */
    critical_section_of_A;
    up(&mutex);      /* leave CS */
    ...
  }
}
```

```
void process_B(void)
{
  while (TRUE) {
    ...
    down(&mutex);  /* enter CS */
    critical_section_of_B;
    up(&mutex);      /* leave CS */
    ...
  }
}
```

Producent-konzument pomocí semaforů

```
#define N 100                       /* number slots in buffer */

semaphore mutex = 1;                /* guards critical section (CS)*/
semaphore empty = N;                /* counts empty slots*/
semaphore full  = 0;                /* counts full slots*/
```

```
void producer(void) {
itemtype item;
while (TRUE) {
  produce_item(&item);
  down(&empty);
  down(&mutex);     /* enter CS*/
  enter_item(item);
  up(&mutex);       /* leave CS*/
  up(&full);
 }
}
```

```
void consumer(void) {
itemtype item;
while (TRUE) {
  down(&full);
  down(&mutex);       /* enter CS */
  remove_item(&item);
  up(&mutex);         /* leave CS */
  up(&empty);
  consume_item(item);
  }
}
```

Monitory

- **Problém se semafory**
  - Pokud například zapomenete použít operaci **`up()`** na konci kritické sekce, potom procesy čekající ve frontě budou čekat navždy (**uváznutí**).

- **Řešení**
  - Vyšší úroveň synchronizace se nazývá **monitory**.
  - Monitor je **konstrukce vyšších programovacích jazyků**, která nabízí stejné možnosti jako semafor.
  - Pouze **jeden proces může být prováděn** uvnitř monitoru v jednom okamžiku.
  - Monitory byly implementovány například v **Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3, Java, .NET,…**

Monitory (2)

- **Monitor**
  - Množina procedur, proměnných a datových struktur, které jsou seskupeny dohromady ve speciálním modulu/objektu.
  - **Pouze jeden proces může být aktivní v daném monitoru v jednom okamžiku.**
  - Překladač (nikoliv programátor) se stará o vzájemné vyloučení uvnitř monitoru.

- **Podmíněné proměnné**
  - Předdefinovaný datový typ, který umožní pozastavit a opět spustit běžící proces.

- **Operace**
  - `Wait(c)`: Pozastaví volající proces na podmíněné proměnné c.
  - `Signal(c)`: Spustí některý z pozastavených procesů na podmíněné proměnné c.

```
monitor Buffer
begin_monitor
  var Buff : array[0..N-1] of ItemType;
      count: integer;
      Full, Empty : condition;

  procedure Insert(E : ItemType);
  begin
      if count = N then Wait(Full);
      insert_item(E);
      count := count + 1;
      if count = 1 then Signal(Empty);
  end;

  procedure Remove(var E : ItemType);
  begin
      if count = 0 then Wait(Empty);
      remove_item(E);
      count := count - 1;
      if count = N - 1 then Signal(Full);
  end;

  count:=0;
end_monitor;
```

Příklad: Problém producenta a konzumenta řešený pomocí monitorů (v Concurrent Pascalu).

```
procedure Producer;
begin
      while true do
      begin
            item := produce_item(item);
            Buffer.Insert(item);
      end
end;
```

```
procedure Consumer;
begin
      while true do
      begin
            Buffer.Remove(item);
            consume_item(item);
      end
end;
```

Zasílání zpráv

- Dvě základní operace:
  - `send(destination,&message)`,
  - `receive(source,&message)`.

- **Synchronizace**
  - Blokující **`send`**, blokující **`receive`** (rendezvous).
  - Neblokující **`send`**, blokující **`receive`**.
  - Neblokující **`send`**, neblokující **`receive`** + test příchozích zpráv.

- **Adresování**
  - **Přímé**: zpráva je uložena přímo do prostoru daného příjemce.
  - **Nepřímé**: zpráva je uložena dočasně do sdílené datové struktury (mailbox). To umožňuje lépe implementovat kolektivní komunikační algoritmy (one-to-one, one-to-many, many-to-many).

- **Příklad**: MPI (Message-Passing Interface).

Příklad: Zasílání zpráv

Problém producenta a konzumenta řešený pomocí zasílání zpráv.

- Typ operací:
  `send()` je neblokující.
  `receive()` je blokující.

- Všechny **zprávy mají stejnou velikost**.

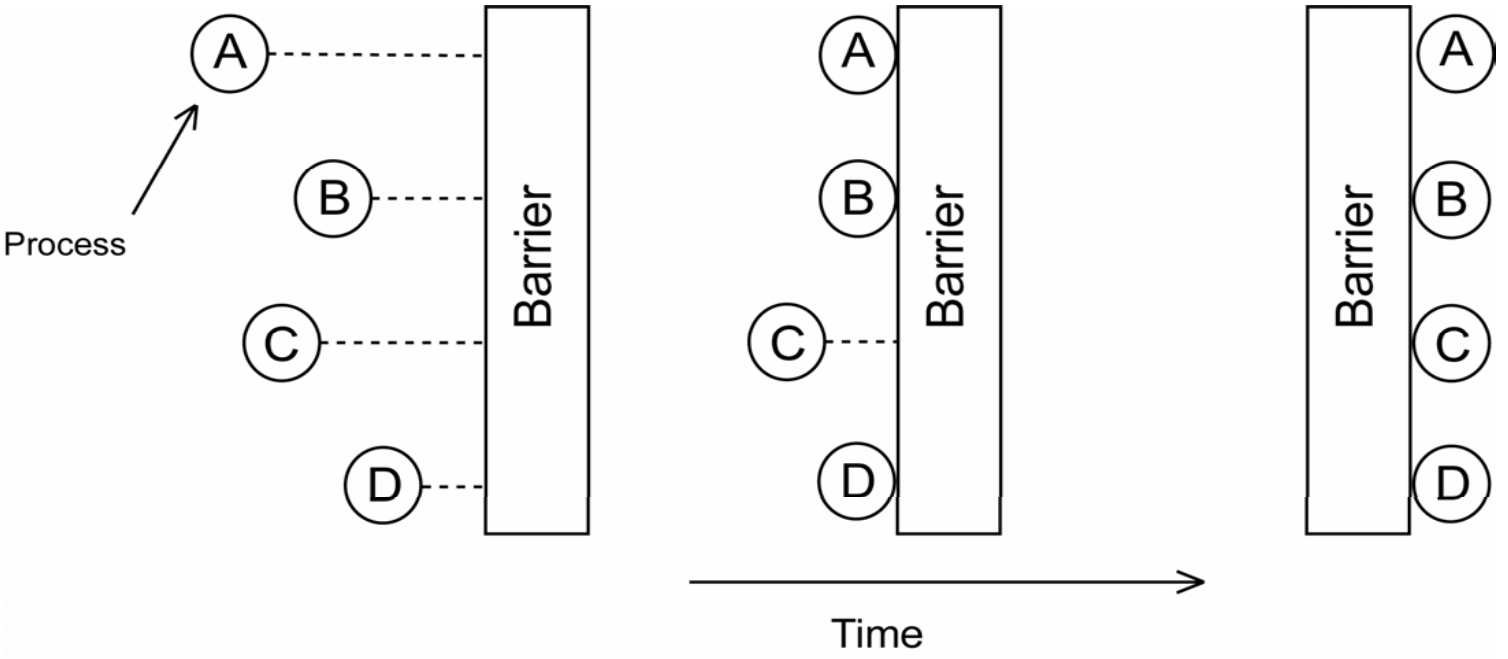
Producent-konzument pomocí zpráv

```
#define N 100                                   /* slots in shared buffer */
void consumer(void) {
  int item, i;
  message m, e;
  for (i = 0; i < N; i++) send(producer, &e); /* send N empties */
  while (TRUE) {
    receive(producer, &m);      /* get message */
    item = extract_item(&m)     /* extract item from message */
    send(producer,&e);          /* send back empty reply */
    consume_item(item); }
}
```

```
void producer(void) {
  int item;
  message m, e;
  while (TRUE) {
    item = produce_item();
    receive(consumer,&e);       /* wait for an empty */
    build_message(&m,item);     /* construct a message */
    send(consumer,&m);          /* send to consumer */ }
}
```

Bariéry

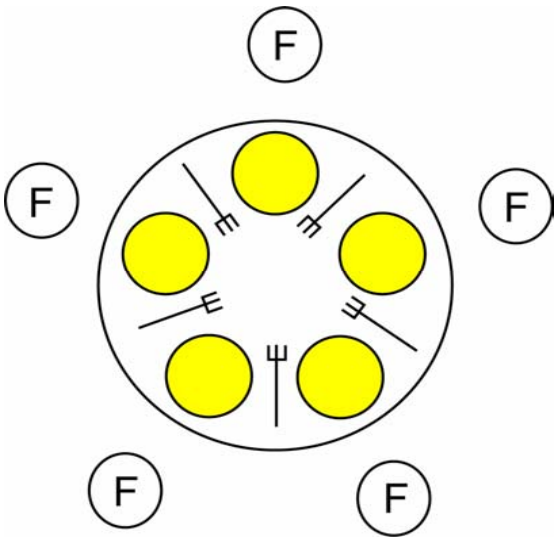
- Pro bariéru definujeme **minimální počet procesů N**, které ji mohou prolomit.
- Když proces dojde k bariéře, tak je zablokován do té doby dokud všech N procesů nedorazí k bariéře.



Večeřící filosofové

- Model procesů, které soutěží o výlučný přístup k omezenému počtu prostředků.

- *N* filozofů sedí kolem kulatého stolu a každý z nich buď přemýšlí nebo jí . K jídlu potřebuje současně levou a pravou vidličku.



Naivní řešení:

```
void philosopher(int i){
    while (TRUE){
        think();
        take_fork(i);
        take_fork((i+1)% N);
        eat();
        put_fork(i);
        put_fork((i+1) % N);
    }
}
```

Večeřící filosofové (2)

- **Naivní řešení může selhat**
  - Všichni filozofové vezmou levou vidličku současně a potom budou čekat až se uvolní pravá vidlička

    => **uváznutí**(budou čekat donekonečna).

- **Vylepšené řešení**
  - Pokud není pravá vidlička k dispozici, filozof vrátí již alokovanou levou vidličku zpět na stůl a pokus o jídlo zopakuje později.

- **Vylepšené řešení může také selhat**
  - Všichni filozofové vezmou levou vidličku.
  - Pak se pokusí vzít pravou vidličku, ale protože není volná, vrátí levou vidličku zpět.
  - Toto budou donekonečna opakovat.

    => **stárnutí** (sice nečekají, ale donekonečna provádí to samé a nikdy se jim to nepovede dokončit)

Večeřící filosofové (3)

```
semaphore mutex=1;

void philosopher(int i){
    while(TRUE){
        think();
        down(&mutex);
            take_fork(i);
            take_fork((i+1) % N);
            eat();
            put_fork(i);
            put_fork((i+1 % N);
        up(&mutex);
    }
}
```

- **Nevýhoda:** pouze jeden filozof může jíst i když by mohlo jíst současně více filozofů.

Večeřící filosofové - řešení

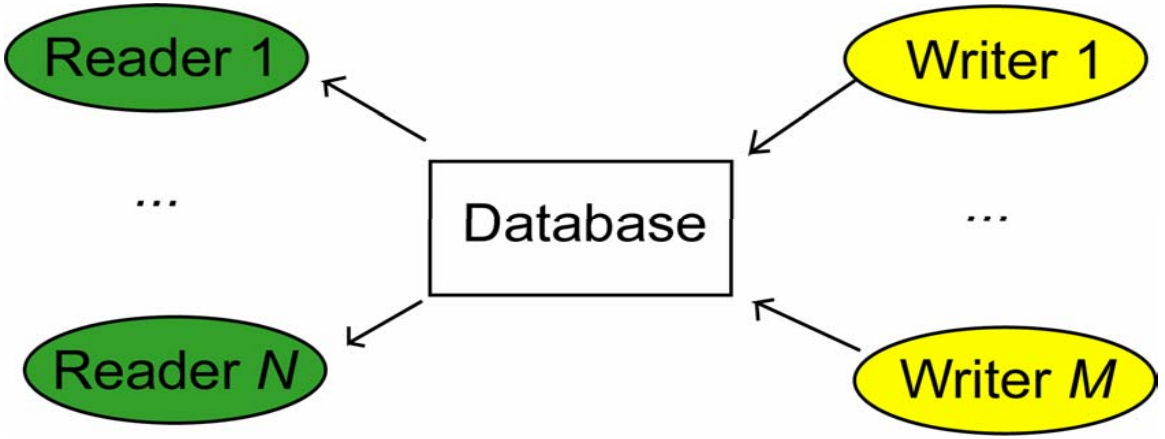
```
#define N 5
#define LEFT ((i+N-1) % N)
#define RIGHT ((i+1) % N)
enum stat(thinking, hungry, eating);
enum stat state[N];
semaphore mutex=1;
semaphore s[N];  # initially set to 0

void philosopher(int i){
        while (TRUE) {
                think();
                take_forks(i);
                eat();
                put_forks(i); }
}
void take_forks(int i) {
        down(&mutex);
          state[i] = hungry;
          test(i);
        up(&mutex);
        down(&s[i]);
}
```

```
void put_forks(int i)
{
        down(&mutex);
          state[i] = thinking;
          test(LEFT);
          test(RIGHT);
        up(&mutex);
}
void test(int i)
{
        if (state[i] == hungry &&
                state[LEFT] != eating &&
                state[RIGHT] != eating)
                {
                        state[i] = eating;
                        up(&s[i]);
                }
}
```

Čtenáři a písaři

- Model procesů, které přistupují do společné databáze.

- Více čtenářů může číst současně data pokud žádný písař nemodifikuje data v databázi.
- Pouze jeden písař může modifikovat data v databázi v jednom okamžiku.



Čtenáři a písaři - řešení

```
int rc = 0;                    /* readers counter */
semaphore mutex = 1;
semaphore db = 1;              /* access to database */
```

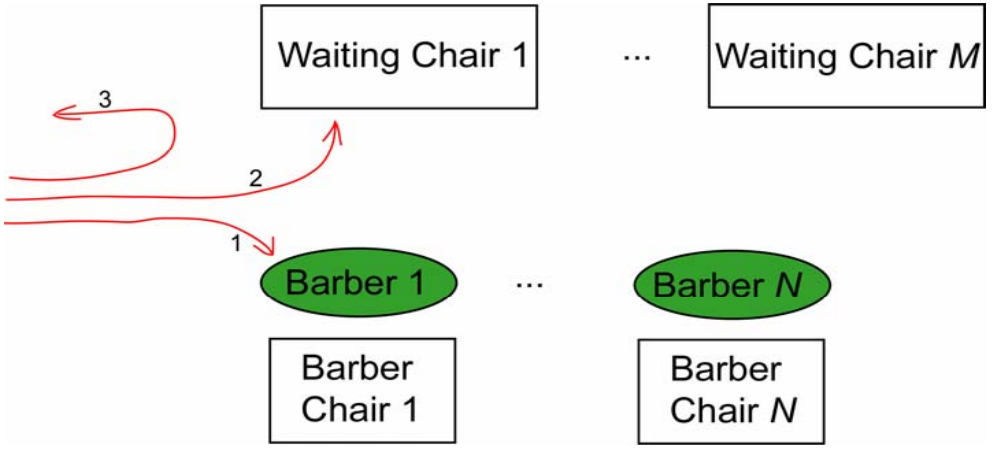
```
void reader(void){
   while(TRUE){
      down(&mutex);
        rc = rc + 1;
        if (rc == 1) down(&db);
      up(&mutex);
      read_data_base(); /* crit. section */
      down(&mutex);
        rc = rc - 1;
        if (rc == 0) up(&db);
      up(&mutex);
      use_data_read(); /* non crit. sect.*/
   }
}
```

```
void writer(void){
    while(TRUE) {
        think_up_data();
        down(&db);
          write_data_data();
        up(&db);
    }
}
```

Spící holiči

- V holičství je *N* holičů (barber), *N* holících křesel (barber chair) a *M* čekacích křesel (waiting chair) pro zákazníky.

- Pokud není žádný zákazník v holičství, holič sedne do holícího křesla a usne.
- Pokud přijde zákazník, potom
  1. pokud nějaký holič spí, tak ho probudí a nechá se ostříhat,
  2. jinak si sedne do křesla a čeká (pokud nějaké je volné),
  3. jinak opustí holičství.



Spící holiči - řešení

```
#define CHAIRS 5
int waiting = 0;                /* customers are waiting (not being cut) */
semaphore mutex = 1;            /* for mutual exclusion                  */
semaphore customers = 0;        /* # of customers waiting for service    */
semaphore barbers = 0;          /* # of barbers waiting for customers    */
```

```
void Barber(void){
    while (TRUE) {
        down(&customers);
        down(&mutex)
          waiting = waiting - 1;
          up(&barbers);
        up(&mutex);
        cut_hair();
    }
}
```

```
void Customer(void){
        down(&mutex);
        if (waiting < CHAIRS) {
                waiting = waiting + 1;
                up(&customes);
                up(&mutex);
                down(&barbers);
                get_haircut();
        } else {
                up(&mutex);
        }
}
```

Operační systémy

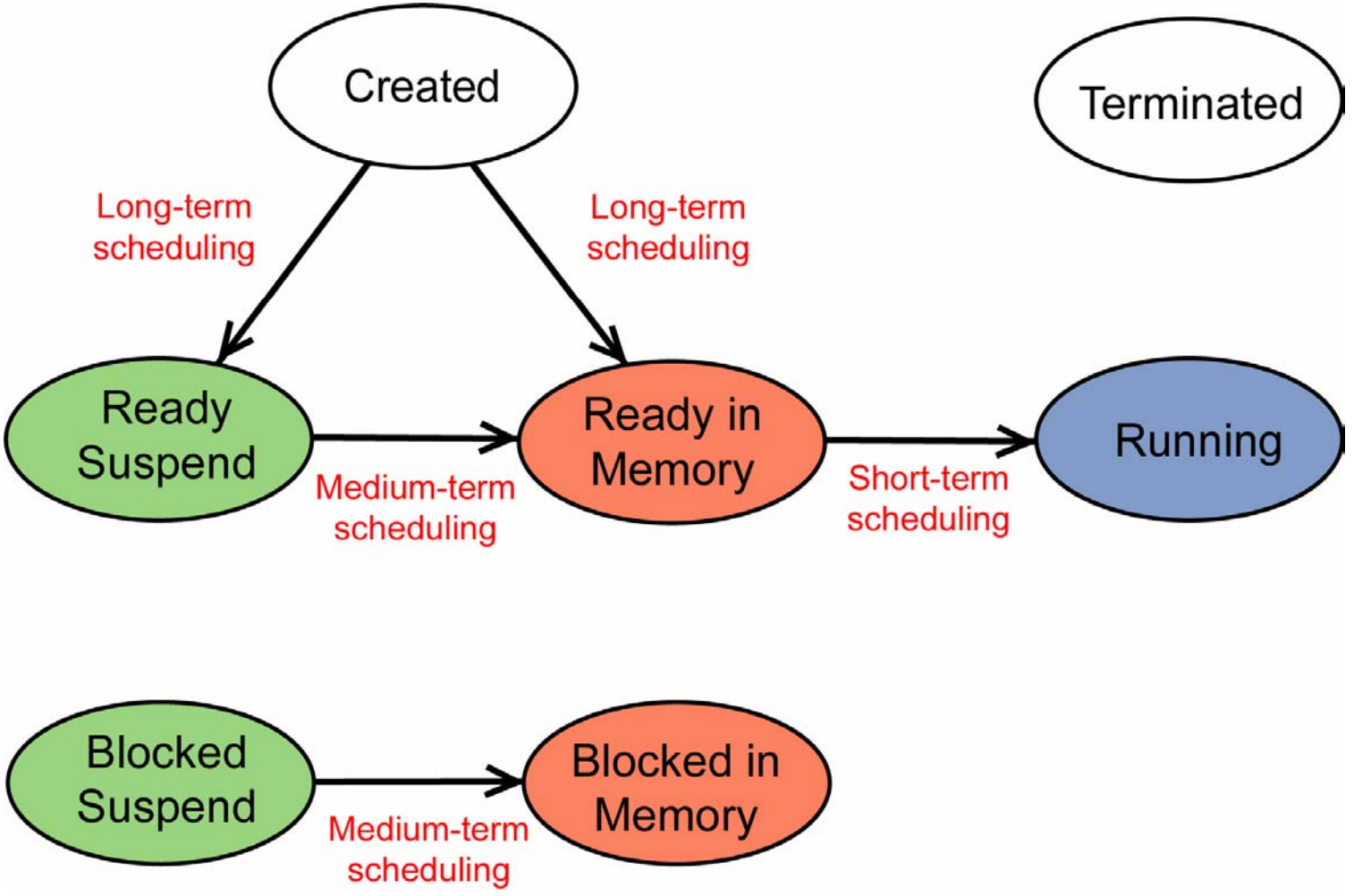
Přednáška 3: Plánování procesů a vláken

Plánovací algoritmy

- Určují, který z čekajících procesů/vláken bude pokračovat.

- **Typy plánování**
  - **dlouhodobé (long-term scheduling)**
    - určuje, které programy budou zpracovány systémem
  - **střednědobé (medium-term scheduling)**
    - součást odkládací funkce
  - **krátkodobé (short-term scheduling)**
    - při přerušení od časovače nebo V/V zařízení, systémové volání, signály,..
  - **V/V (I/O scheduling)**
    - určuje, který V/V požadavek bude obsloužen volným V/V zařízením

Typy plánování



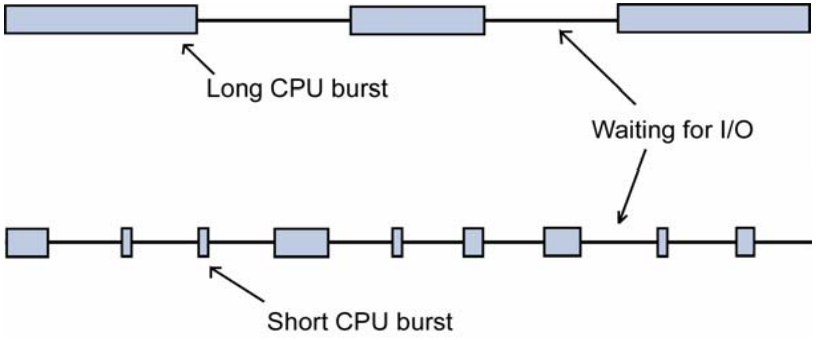
Kritéria krátkodobého plánování

- **Uživatelské hledisko**
  - **doba zpracování (turnaround time)**
    - doba, která uplyne od spuštění do ukončení procesu
  - **doba odezvy (response time)**
    - doba, která uplyne od okamžiku zadání požadavku do doby první reakce
  - **dosažení meze (deadlines)**
    - zaručení ukončení procesu do dané meze
  - **předvídatelnost (predictability)**
    - proces by měl pokaždé běžet „stejně“ dlouho bez ohledu na zatížení systému

Kritéria krátkodobého plánování (2)

- **Systémové hledisko**
  - **propustnost (throughput)**
    - počet procesů dokončených za jednotku času
  - **vyžití procesoru (processor utilization)**
  - **spravedlivost (fairness)**
    - zamezit hladovění (starvation)
  - **prosazení priorit (enforcing priorities)**
    - možnost uplatnit zvolenou plánovací strategii
  - **vyvážení V/V (balancing resources)**
    - snaha udržet V/V prostředky maximálně vytížené

Typy procesů/vláken



- **Orientované na CPU (CPU-bound processes)**
  - CPU využíváno po dlouhou dobu, málo časté čekání na V/V.
- **Orientované na V/V (I/O-bound processes)**
  - CPU využito po krátkou dobu, velmi časté čekání na V/V.

- Klíčový faktor je doba využití CPU
- Procesy mají tendenci stále více využívat V/V prostředky než CPU, ale CPU se zdokonalují rychleji než V/V prostředky.
  - musíme **vytěžovat V/V prostředky co nejvíce**
  - když proces orientovaný na V/V a chce CPU, měl by ho co nejdříve dostat

Kdy plánujeme?

- **Vytvoření nového procesu**
- **Ukončení běžícího procesu**
- **Běžící proces je zablokován z důvodu provádění systémového volání**
  - Důvod zablokování může hrát roli při plánování (např. proces A čeká až proces B opustí kritickou sekci).
  - Většinou ale plánovač nemá k dispozici tuto informaci.
- **Přerušení od V/V zařízení**
  - Vybereme nějaký „ready“ proces nebo „blocked“ proces čekající na V/V?
- **Přerušení od časovače**
  - Plánovací rozhodnutí při každém přerušení od časovače nebo při *k*-tém přerušení od časovače.

Strategie plánování

- **Plánování s předbíháním (preemptive scheduling)**
  - Běžící proces je zablokován automaticky po uplynutí časového kvanta.
  - Jediná možná strategie v RT-systémech a více uživatelských systémech.

- **Plánování bez předbíhání (nonpreemtive scheduling)**
  - Proces běží dokud nepožádá o nějakou službu jádro nebo neskončí.
  - Proces může blokovat systém a musí „spolupracovat“ pouze když žádá službu jádra.
  - Používalo se v dávkových systémech.
  - Používá se v některých systémech pro speciální třídy procesů/vláken.

Plánování v dávkových systémech

- **First-Come First-Served (FCFS)**
  - Plánování bez předbíhání pomocí jedné FIFO fronty procesů.
  - Když je běžící proces zablokován, první proces ve frontě bude pokračovat.
  - Když se zablokovaný proces stane opět „ready“, je zařazen na konec fronty.

  - **Výhody:** jednoduché na pochopení i implementování.
  - **Nevýhody:** FCFS může zpomalit procesy orientované na V/V.

Plánování v dávkových systémech (2)

- **Shortest Job First (SJF)**
  - Plánování bez předbíhání, které předpokládá, že **doba výpočtu je známa předem**.
  - Plánovač spouští nejdříve procesy s nejmenší dobou výpočtu.
  - SJF minimalizuje průměrnou dobu zpracování.

Proces	A	B	C	D
Čas výpočtu [min]	8	4	4	4
Pořadí zpracování A,B,C,D				
Doba zpracování	8	12	16	20
Průměrná doba zpracování 14 min				
Pořadí zpracování D,C,B,A				
Doba zpracování	20	12	8	4
Průměrná doba zpracování 11 min				

Plánování v dávkových systémech (3)

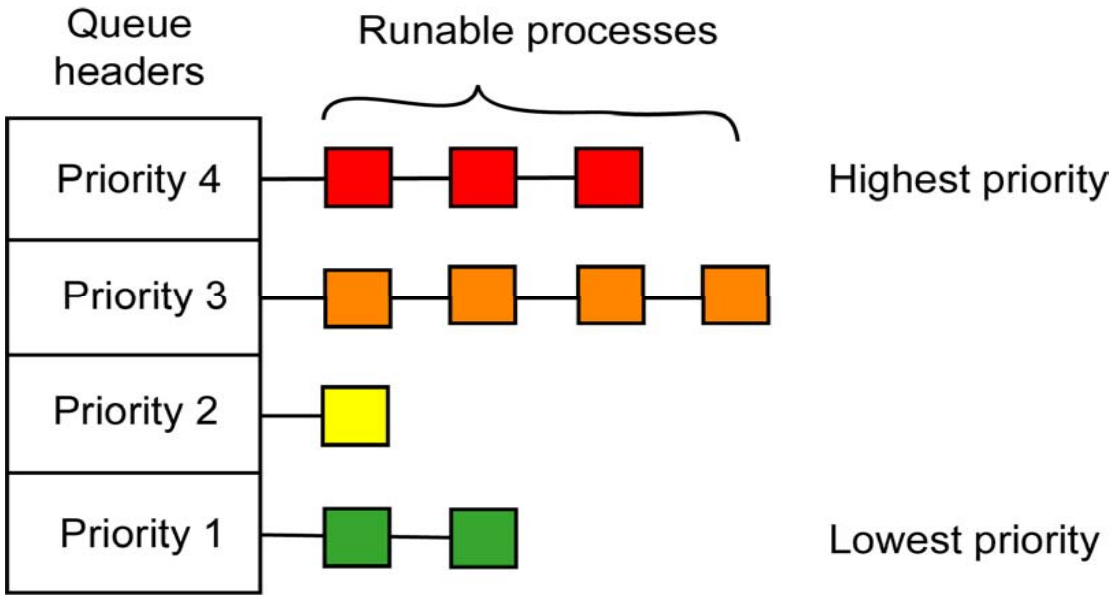
- **Shortest Remaining Time Next (SRT)**
  - Verze SJF pro plánování s předbíháním.
  - Plánovač přidělí CPU procesu s nejmenším časem výpočtu.
  - Když přijde požadavek na spuštění nového procesu, plánovač porovná časy výpočtu nového procesu a aktuálního procesu.
  - Proces s menším časem výpočtu získá CPU.
  - **Nevýhoda:** riziko hladovění procesů s velkým časem výpočtu.

Plánování v interaktivních systémech

- **Round-Robin Scheduling (RR)**
  - Plánování s předbíháním.
  - „Ready“ procesy čekají ve frontě FIFO.
  - Každý proces dostane přiděleno CPU na **časové kvantum**.
  - Po uplynutí časového kvanta, běžící proces je pozastaven a uložen na konec FIFO fronty, první proces ve FIFO frontě dostane přiděleno CPU.
  - Délka časového kvanta je důležitý parametr:
    - Krátké čas. kvantum ⇒ nízká efektivita CPU.
    - Dlouhé čas. kvantum ⇒ špatná doba odezvy.
    - Rozumný kompromis je 20-50ms.

Plánování v interaktivních systémech (2)

- **Priority Scheduling (PS)**
  - Každý proces má přidělenu **prioritu**.
  - „Ready“ procesy se stejnou prioritou jsou seskupeny do **prioritních tříd**.
  - **CPU je přiřazováno procesům z nejvyšší prioritní třídy**.
  - Procesy z nejvyšší prioritní třídy se střídají **metodou round-robin**.



Plánování v interaktivních systémech (3)

- **Varianty prioritního plánování**
  - **bez předbíhání**
  - **s předbíháním**
    - **statická priorita**
    - **dynamická priorita**

    - **fixní časové kvantum** pro všechny prioritní třídy
    - **různě velká časová kvanta** pro různé prioritní třídy

- **Problém**
  - **hladovění (starvation)**
  - Proces s nízkou prioritou nedostane CPU.

- **Řešení**
  - procesu, který čeká déle než je stanovený limit, je **dočasně zvýšena priorita**

Příklad: zvýhodnění procesů orientovaných na V/V

- Procesy orientované na V/V stráví většinu času čekáním na dokončení V/V operací.
- Procesy orientované na V/V by měly dostat C/PU co nejdříve, aby mohly opět zahájit V/V operaci.
- **Každému procesu nastavíme dynamickou prioritu $P=t_q/t_u$**
  - **t_q** je časové kvantum a
  - **t_u** je čas skutečně strávený na CPU.
- Např. proces, který využil 2ms ze 100ms čas. kvanta dostane prioritu 50. Zatím co proces, který běžel 50ms (ze 100ms čas. kvanta) dostane prioritu 2.

Příklad: redukování režie na přepínání kontextu

- Operační systém CTSS (MQ)
  - **přepínání kontextu bylo velmi pomalé**, neboť počítač (IBM 7094) mohl mít v paměti pouze jeden proces.
  - Proto bylo vhodné nastavit velké časové kvantum, abychom **redukovali režii na přepínání kontextu**.

- **Proces v nejvyšší třídě běží po dobu jednoho čas. kvanta. Proces v následující třídě poběží po dobu dvou čas. kvant, …**

- **Pokud proces využije celé čas. kvantum, bude přesunut do následující nižší třídy.**

- Např. proces, jehož doba výpočtu je 100q, dostane 1q, 2q, 4q, 8q, 16q, 32q, 64q. Z posledního čas. intervalu (64q) využije pouze 37q.
  - v MQ dojde pouze k 7 přepnutím kontextu
  - v RR by došlo ke 100 přepnutím kontext.

Příklad: Shortest Process Next (SPN)

- Modifikace SJF pro interaktivní OS.
- Čas výpočtu procesu se odhaduje na základě minulého chování procesu.

- Proces s nejmenším odhadem času výpočtu dostane CPU.
- Odhad:

$$T_0 \ldots\ldots\ldots\ldots\ldots \text{odhad},$$

$$T_1 \ldots\ldots\ldots\ldots\ldots \text{čas následujícího behu}$$

$$kT_0 + (1-k)T_1 \ldots \text{nový odhad } (0 \le k \le 1).$$

- Pro k=1/2, dostaneme

$$T_0, \quad T_0/2 + T_1/2, \quad T_0/4 + T_1/4 + T_2/2, \quad T_0/8 + T_1/8 + T_2/4 + T_3/2$$

- Využívá se techniky **stárnutí (aging).**

Windows XP – plánování vláken

- **Plánování CPU je po vláknech** a je implementováno v jádru.

- Používá **prioritní plánování s předbíháním**

  – CPU je přidělováno **„ready“ vláknům s nejvyšší prioritou metodou „round robin“**

  – CPU může vlákno využívat po **dobu časového kvanta**, pokud ho nepřeruší vlákno s vyšší prioritou

  – **časové kvantum může být různé** pro různé systémy/vlákna

Windows XP – plánování vláken (2)

- **Jádro: rozlišuje 32 priorit**
  - real-time úrovně (16-31)
  - dynamické úrovně (1-15)
  - systémová úroveň (0)
- **Windows API**
  - Základní priority procesů (Realtime, High, Above Normal, Normal, Below Normal, Idle)
  - Základní priority vláken (Time critical, Highest, Above Normal, Normal, Below Normal, Lowest, Idle)

		Windows API process priorities					
Windows API thread priorities		Realtime	High	Above Normal	Normal	Below Normal	Idle
	Time critical	31	15	15	15	15	15
	Highest	26	15	12	10	8	6
	Above Normal	25	14	11	9	7	5
	Normal	24	13	10	8	6	4
	Below Normal	23	12	9	7	5	3
	Lowest	22	11	8	6	4	3
	Idle	16	1	1	1	1	1

Windows XP – plánování vláken (3)

- **Proces**
  - **základní prioritu** (base priority), viz. předchozí tabulka
  - implicitně se dědí od rodiče
  - při spuštění procesu (např. funkcí `CreateProcess()` nebo příkazem `start /úroveň program`)
  - po spuštění (např. funkcí `SetPriorityClass()` nebo aplikací `Task Manager`)

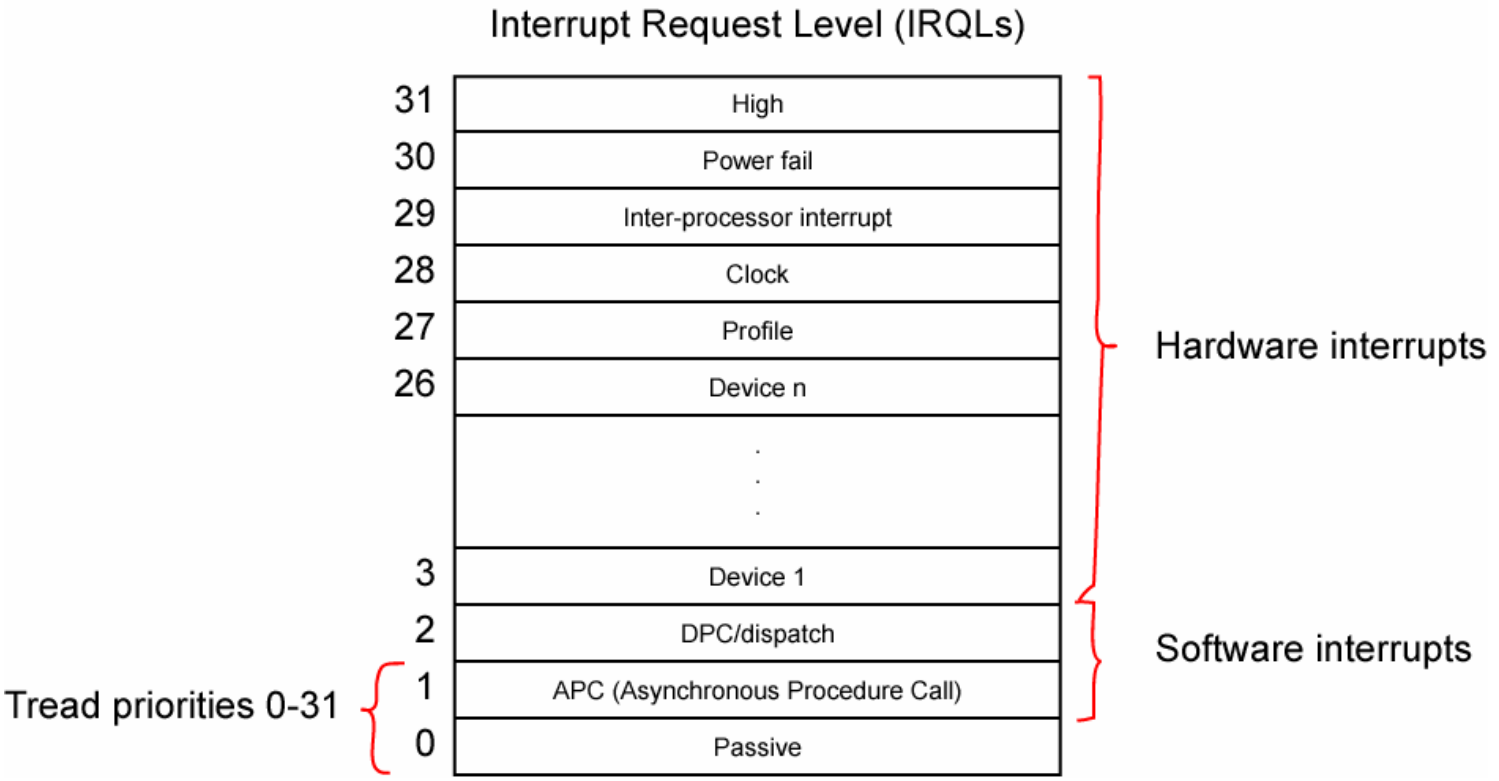
- **Vlákno**
  - **základní priorita** (base priority), dědí se od základní priority procesu
  - **aktuální priorita** (dynamic priority)

- **Plánování se děje na základě aktuální priority vlákna**.

- Jádro může dočasně modifikovat aktuální prioritu vlákna (kromě real-time úrovně 16-31).

Windows XP – plánování vláken (4)

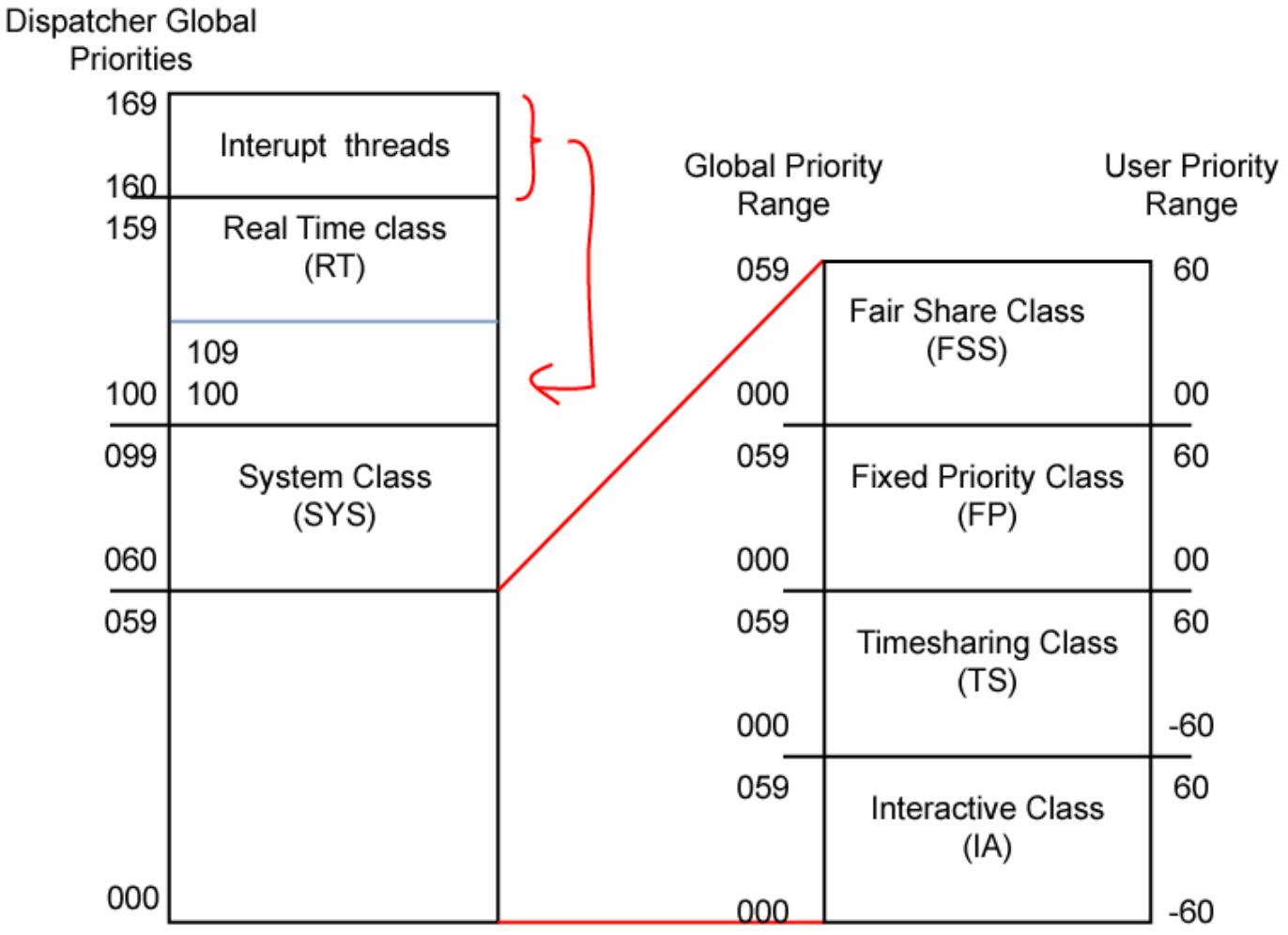
- Interrupt Levels x Priority levels



Windows XP – plánování vláken (5)

- **Zobrazení informací o procesech/vláknech**
  - aplikace **Správce úloh** (CTRL-ALT-DEL)
  - příkaz **tasklist.exe**
  - další nástroje z **`www.sysinternals.com`**
    - aplikace **Process explorer (`ProcExp.exe`)**

Solaris – plánování vláken



Solaris – plánování vláken (2)

- **Plánovací třídy**:
  - **Timesharing (TS)**
    - normální uživ. procesy,
    - prioritní plánování s předbíháním, různě velká čas. kvanta
    - procesy orientované na CPU mají prioritu nižší než procesy orientované na V/V
    - procesy, které nedostaly CPU během určitého času, budou mít zvýšenou prioritu.
  - **Interactive (IA)**
    - podobné jako TS, ale priorita procesů v aktivním okně se rychleji zvyšuje.
  - **Fair Share (FSS)**
    - pro vlákna, která jsou součástí projektu
    - priorita je přidělena, měněna podle limitů nastavených na projekt
  - **Fixed Priority (FX)**
    - přidělená priorita není měněna jádrem
    - prioritní plánování s předbíháním, různě velká čas. kvanta

Solaris – plánování vláken (3)

- **Real Time (RT)**
  - RT procesy,
  - nejvyšší priorita (pouze obsluha přerušení má vyšší)
  - plánování s předbíháním,
  - fixní priorita.
- **System (SYS)**
  - systémové vlákna,
  - prioritní plánování bez předbíhání,
  - fixní priorita.

Solaris – plánování procesů (4)

- Informace o parametrech plánování

dispadmin -c TS -g

# ts_quantum	ts_tqexp	ts_slpret	ts_maxwait	ts_lwait	PRIORITY LEVEL
#					
500	0	10	5	10	# 0
500	0	11	5	11	# 1
500	1	12	5	12	# 2
500	1	13	5	13	# 3
500	2	14	5	14	# 4
500	2	15	5	15	# 5
450	3	16	5	16	# 6
450	3	17	5	17	# 7
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
50	48	59	5	59	# 58
50	49	59	5	59	# 59

Solaris – plánování procesů (5)

- **ts_quantum**
  - Aktuální časové kvantum pro danou prioritu.
- **ts_tqexp**
  - Nová priorita pro proces, který využil celé čas. kvantum.
- **ts_slpret**
  - Nové priorita pro proces, který nevyužil celé čas. kvantum.
- **ts_maxwait**
  - Pokud proces nedostal během tohoto intervalu CPU, tak bude mít novou prioritu **ts_lwait.**
- **PRIORITY LEVEL**
  - Aktuální priorita procesu.

Solaris – plánování procesů (6)

- Jak změnit plánovací třídu procesu?

  dispadmin

- Jak změnit prioritu procesu?

  priocntl -s -c třída -t časové_kvantum -p priorita program

  nice +10 příkaz

Operační systémy

Přednáška 6: Uváznutí procesů/vláken

Výpočetní prostředky

- **Výpočetní prostředek**
  - **Hardwarové zařízení** (např. pásková mechanika, tiskárna, CD vypalovačka,…).
  - **Informace** (např. položka v databázi, řádek v systémové interní tabulce, soubor,…).

- **Výlučný přístup** k prostředkům
  - Většina výpočetních prostředků může být **v jednom okamžiku používána pouze jedním procesem** (např. tiskárna, páska,…).

- Většina procesů potřebuje **výlučný přístup k více prostředkům.**

- **Dva typy výpočetních prostředků**
  - **Odnímatelné (Preemptable)**: mohou být odebrány procesu bez rizika dalších problémů (např.odložení procesu z fyzické paměti na disk).
  - **Neodnímatelné (Nonpreemptable)**: nemohou být odebrány bez rizika (např. CD-ROM vypalovačka).

Výpočetní prostředky (2)

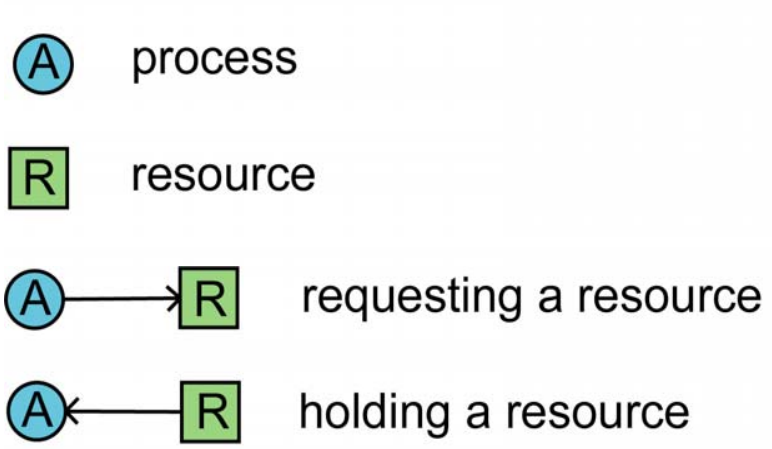
- **Sekvence kroků** při použití prostředku:
  1. **Žádost** o prostředek.
  2. **Použití** prostředku.
  3. **Uvolnění** prostředku.

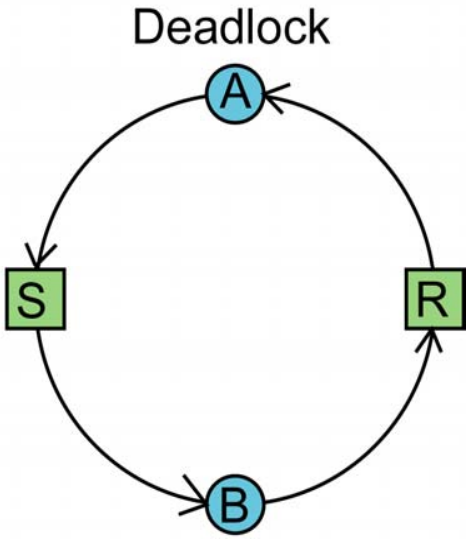
- **Žádný prostředek není dostupný**
  - Žádající proces bude **automaticky zablokován** prostřednictvím OS a probuzen až daný prostředek bude k dispozici.
  - Žádost skončí s chybou a
    - žádající proces se **ukončí** ihned nebo po několika neúspěšných pokusech
    - žádající proces se bude **čekat na prostředek** (aktivně/pasivně) prostředek.

- **Definice uváznutí (deadlock):** Množina procesů uvázne pokud každý proces v množině čeká na nějakou událost, kterou může vyvolat pouze některý proces z množiny.

Modelování uváznutí

- Uváznutí můžeme modelovat pomocí **alokačního grafu**.





- **Každá smyčka v grafu představuje uváznutí** (procesy ve smyčce čekají a nemohou pokračovat).

Příklad: uváznutí

proces A	**proces B**	**proces C**
Žádost o R	Žádost o S	Žádost o T
Žádost o S	Žádost o T	Žádost o R
uvolnění R	uvolnění S	uvolnění T
uvolnění S	uvolnění T	uvolnění R

Alokace s uváznutím:

A: Žádost o R
B: Žádost o S
C: Žádost o T
A: Žádost o S ⇒ proces je uspán
B: Žádost o T ⇒ proces je uspán
C: Žádost o R ⇒ **uváznutí !!!**

Alokace bez uváznutí:

A: Žádost o R
C: Žádost o T
A: Žádost o S
C: Žádost o R ⇒ proces je uspán
A: uvolnění S . . . **bez uváznutí!!!**

Coffmanovy podmínky

- Nutné podmínky uváznutí
  1. **Vzájemné vyloučení.** Každý prostředek je buď přidělen právě jednomu procesu a nebo je volný (prostředek nemůže být sdílen více procesy).

  2. **Podmínka „drž a čekej“.** Proces, který má již přiděleny nějaké prostředky, může žádat o další prostředky (proces může žádat o prostředky postupně).

  3. **Podmínka neodnímatelnosti.** Prostředek, který byl již přidělen nějakému procesu, nemůže mu být násilím odebrán. Musí být dobrovolně uvolněn daným procesem.

  4. **Podmínka cyklického čekání.** Musí existovat smyčka dvou nebo více procesů, ve které každý proces čeká na prostředek přidělený následujícímu procesu ve smyčce.

- **Pokud aspoň jedna z podmínek není splněna, nemůže dojít k uváznutí.**

Strategie řešení uváznutí

1. **Pštrosí algoritmus.** Úplné ignorování celého problému.

2. **Detekce a zotavení.** K uváznutí může dojít, ale pak je detekováno a odstraněno.

3. **Dynamické zamezení vzniku uváznutí** pomocí pečlivé alokace prostředků.

4. **Prevence** pomocí nesplnění aspoň jedné z Coffmanových podmínek.

Pštrosí algoritmus

- Praktické řešení ve většině univerzálních OS (UNIX, MS windows,…).
- **Uváznutí** se vyskytuje relativně **zřídka**, např. jednou za rok.
- **Hardwarové chyby** a **chyby v OS** se vyskytují **mnohem častěji**, např. každý měsíc (záplaty,…).
- **Zabránění uváznutí je drahé.**
- Např. v Unixu, nejčastěji používané prostředky v jádru jsou položky v tabulce procesů a položky v tabulce i-nodů (mající samozřejmě omezenou kapacitu).
- **Řešení:**
  - Proces se pokusí alokovat prostředek. Pokud není k dispozici, tak skončí s chybou.
  - Administrátor zjistí uvázlé procesy a ukončí je.
- Toto řešení **není přijatelné ve„fault tolerant“ systémech**.

Detekce a zotavení

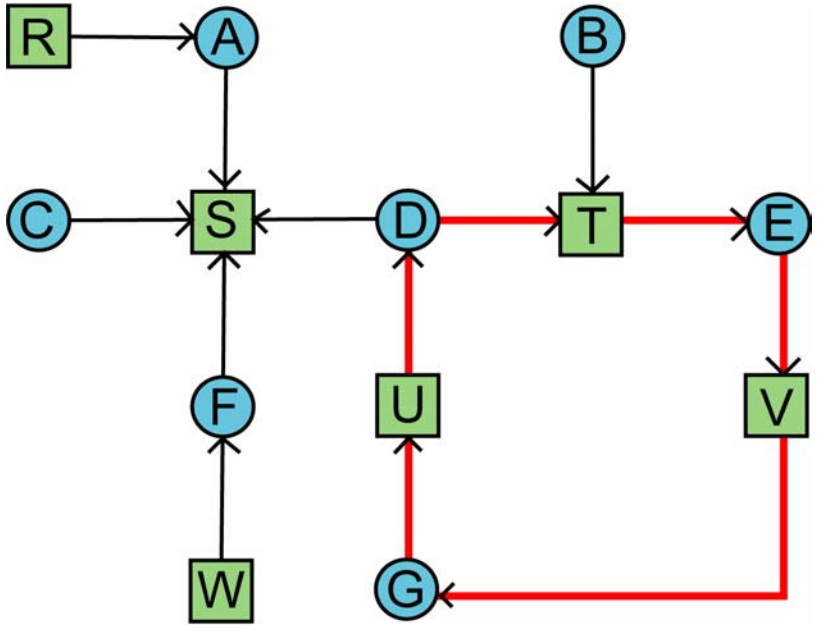
- Systém se **nesnaží předcházet výskytu uváznutí**, tzn. že procesy mohou uváznout.

- V případě výskytu uváznutí se tuto situaci snaží **detekovat** a na základě toho se pokusí **uváznutí odstranit**.

Detekce uváznutí

- Předpokládejme, že existuje pouze **jeden prostředek od každého typu**.

- Sestrojíme **alokační graf**.
  - Jakýkoliv proces, který je součástí smyčky, je uváznutý.
  - Pokud v grafu neexistuje žádná smyčka, tak v systému nedošlo k uváznutí.



Detekce uváznutí (2)

- Předpokládejme **více prostředků od každého typu** (*m*).
- Algoritmus pro detekci mezi *n* procesy:

Exitující prostředky

$$E = (E_1, E_2, E_3, \ldots, E_m)$$

Volné prostředky

$$A = (A_1, A_2, A_3, \ldots, A_m)$$

Již alokované prostředky

$$C = \begin{bmatrix} C_{11} & C_{12} & C_{13} \cdots & C_{1m} \\ \vdots & \vdots & \vdots & \vdots \\ C_{n1} & C_{n2} & C_{n3} \cdots & C_{nm} \end{bmatrix}$$

Ještě požadované prostředky

$$R = \begin{bmatrix} R_{11} & R_{12} & R_{13} \cdots & R_{1m} \\ \vdots & \vdots & \vdots & \vdots \\ R_{n1} & R_{n2} & R_{n3} \cdots & R_{nm} \end{bmatrix}$$

Každý prostředek je již alokován nebo je volný:

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

Detekce uváznutí (3)

- Každý proces je **na začátku neoznačen.**

- **Algoritmus pro detekci uváznutí**:
  1. Najdi neoznačený proces P_i, jehož *i*-tý řádek v R má menší nebo stejné hodnoty jako A.
  2. Pokud takový proces existuje, přičti hodnoty *i*-tého řádku z C k hodnotám v A, označ tento proces a pokračuj bodem 1.
  3. Pokud žádný takový proces neexistuje, algoritmus končí.

- Po skončení algoritmu, **všechny neoznačené procesy jsou uvázlé.**

Příklad: detekce uváznutí

- Máme **3 procesy** a **4 typy prostředků** (např. pásky, plotry, skenery, CR-ROM).
- Aktuální rozdělení je E = (4, 2, 3, 1), A = (2, 1, 0, 0)

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix} \qquad R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

- Třetí proces může být upokojen a potom uvolní své alokované prostředky $\Rightarrow$ A=(2, 2, 2, 0).
- Dále může být uspokojen druhý proces, který také vrátí své prostředky $\Rightarrow$ A=(4, 2, 2, 1).
- Nyní poslední proces může pokračovat $\Rightarrow$ **Bez uváznutí!**

Zotavení z uváznutí

- **Zotavení pomocí odebrání**
  - Dočesné násilné odebrání prostředku a půjčení jinému procesu. Ve většině případů to vyžaduje manuální intervenci, (např. odebrání pásky nebo tiskárny).

- **Zotavení promocí návratu**
  - **Periodicky si ukládáme důležité informace o procesech**, tak abychom později byli schopni je vrátit do předchozích stavů.

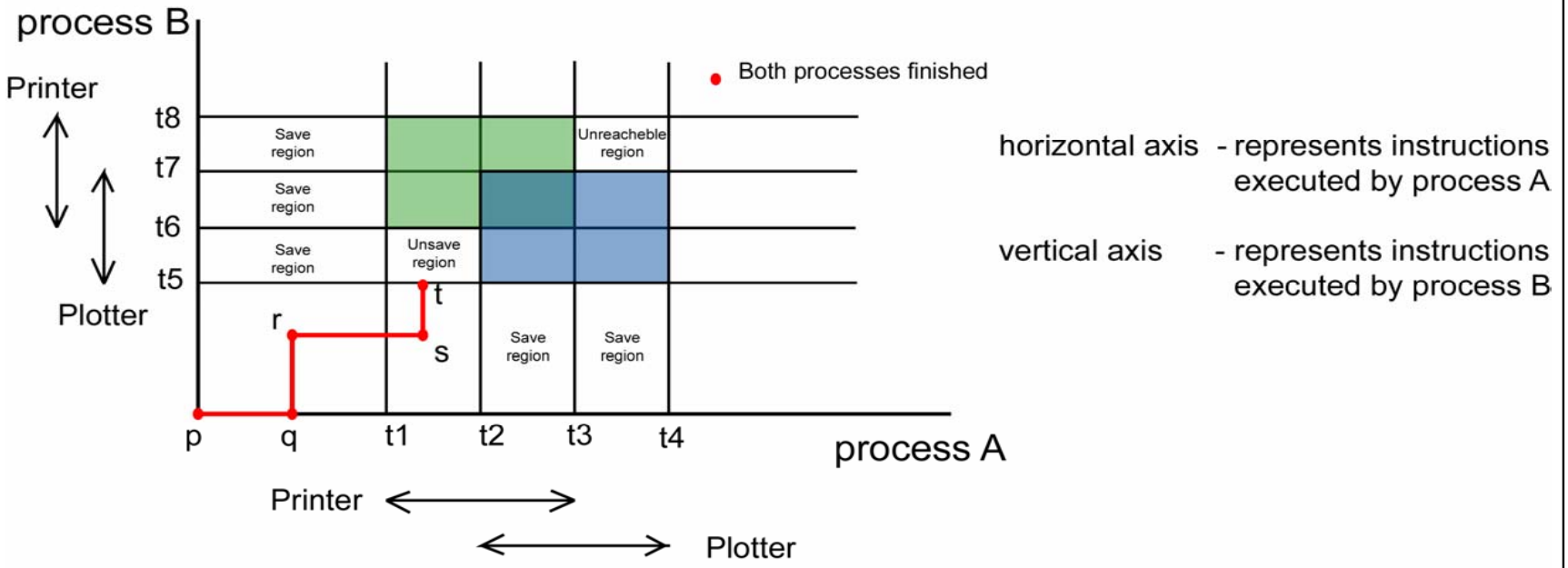
  - Při detekci uváznutí, **proces, který vlastní kritický prostředek, je vrácen zpět** v čase do stavu, kdy daný prostředek ještě nevlastnil. Uvolněný kritický prostředek je přidělen jednomu z uváznutých procesů.

- **Zotavení pomocí ukončení procesů**
  - Ukončíme proces, který je součástí **smyčky** v alokačním grafu.
  - Problém s některými typy procesů (překlad x modifikace databáze).

Zamezení vzniku uváznutí

- Většinou procesy alokují prostředky postupně jeden po druhém.
- V okamžiku žádosti o další prostředek, systém musí rozhodnout, zda přidělení prostředku je **bezpečné** (tzn. že nedojde k uváznutí) či **nebezpečné**.
- Prostředek bude **přidělen pouze pokud to bude bezpečné**.
- Systém se snaží předejít uváznutí **opatrnou alokací prostředků**.
- **Postupový prostor** (postupová cesta) lze použít pro modelování vzniku uváznutí.

Postupový prostor



- **Vstup do barevných obdélníků je zakázán** díky výlučnému přístupu k prostředku (uvnitř barevné oblasti by prostředek byl alokován současně dvěma procesy).
- **Procesy nesmí vstoupit do nebezpečné oblasti (Unsave region).** Vstup do této oblasti vždy skončí uváznutím.
- V bodě *t*, systém musí rozhodnout, zda přidělí ploter procesu B.
- Abychom se vyhnuli uváznutí, B musí být pozastaven.
- Proces B bude pokračovat, až proces A použije a opět uvolní ploter.

Bankéřův algoritmus

- Algoritmus, který předchází vzniku uváznutí (Dijkstra, 1965).

- **Bezpečný stav**
  - Pokud nedošlo k uváznutí a existuje takové alokační pořadí, které zaručuje, že každý proces bude postupně uspokojen a skončí.

- Bankéřův algoritmus kontroluje, zda **přidělení prostředku vede na bezpečný stav**.
  - Pokud **ne**, pak je **žádost odmítnuta**.
  - Pokud **ano**, **prostředek je přidělen**.

- **Nevýhoda:** proces musí dopředu vědět, které prostředky bude během svého života potřebovat.

Příklad: bankéřův algoritmus

- V systému je 10 prostředků stejného typu.
- Procesy A, B, C a D budou dohromady potřebovat 22 prostředků.

Počáteční stav

proces	má	max
A	0	6
B	0	5
C	0	4
D	0	7

Bezpečný stav

proces	má	max
A	1	6
B	1	5
C	2	4
D	4	7

Nebezpečný stav

proces	má	max
A	1	6
B	1	5
C	2	4
D	5	7

- **Bezpečný stav:** (2 volné prostředky), pokud dáme 2 prostředky procesu C, ten po skončení uvolní 4 prostředky a ostatní procesy mohou pokračovat.
- **Nebezpečný stav:** (pouze 1 volný prostředek), žádný proces nemůže pokračovat.

Prevence uváznutí

- Nesplnění aspoň jedné z Coffmanových podmínek

1. **Porušení výlučného přístupu**
   - **Sdílení prostředku** s výlučným přístupem pomocí **virtualizace**.

- **Příklad: sdílení tiskárny**
  - Každý proces pošle svůj požadavek do tiskové fronty.
  - Tiskový démon postupně požadavky z tiskové fronty posílá na fyzickou tiskárnu.
  - Pouze tiskový démon přistupuje přímo k fyzické tiskárně.

- Bohužel toto řešení nelze použít ve všech případech, např. pásky.

Prevence uváznutí (2)

2. **Porušení podmínky „drž a čekej“**
   - Každý proces musí **alokovat všechny požadované prostředky v okamžiku spuštěním** (např. OS/360).
   - **Prostředky nebudou využívány optimálně.**
   - Například:

     Proces si alokuje pásku, 5 minut z ní načítá vstupní data, potom 50 minut provádí výpočet a nakonec výsledek tiskne 5 minut na tiskárnu.
   - Páska i tiskárna jsou alokovány po celých 60 minut.

- Pokud budeme znát požadavky na prostředky v okamžiku spuštění procesu můžeme použít **bankéřův algoritmus.**

Prevence uváznutí (3)

3. Porušení podmínky neodnímatelnosti

– V praxi těžko realizovatelné.

4. Porušení podmínky kruhového čekání

a) Proces může mít přidělen v daném okamžiku pouze jeden prostředek.

b) Proces může alokovat více prostředků, ale pouze v přesně definovaném pořadí.

– Prostředky mají přiřazena čísla (např. ptiskárna 1, ploter 2, páska 3).

– Proces může požádat pouze o prostředek s vyšším číslem než je maximum z již alokovaných prostředků.

– Problém: pro daný počet procesů a prostředků **nemusí existovat vhodné očíslování prostředků**.

Operační systémy

Přednáška 7: Správa paměti I

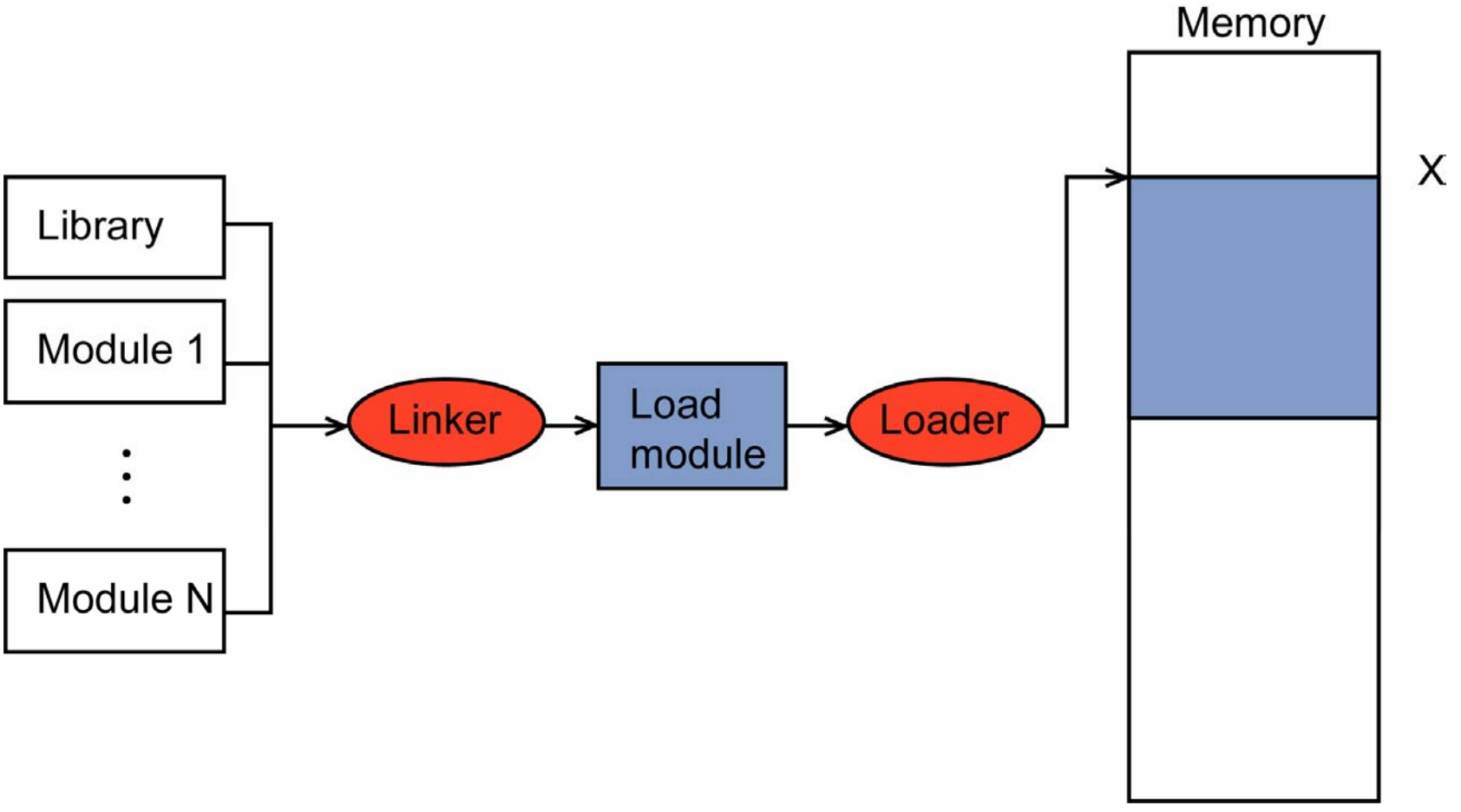
Správa paměti (SP)

- **Memory Management Unit (MMU)**
  - hardware umístěný na CPU čipu
  - např. překládá logické adresy na fyzické adresy,…
- **Memory Manager**
  - software, který je součástí OS
  - udržuje **informaci volné a přidělené** paměti
  - **přiděluje a uvolňuje** paměť
  - zajišťuje **odkládání (swapping)** procesů

Požadavky na SP

- **Přemístění (Relocation)**
  - při kompilaci není většinou známo umístění procesu ve fyzické paměti
  - každý **odkaz do paměti** v programu **musí být přepočítán** podle aktuálního umístění procesu ve fyzické paměti
  - v programu se můžeme **odkazovat na další programy**

Spojování a zavedení programu



Zavádění programu (loading)

- **absolutní zavedení (absolute loading)**
  - každý odkaz do paměti v programu obsahuje absolutní fyzickou adresu
  - program musí být zaveden vždy od dané fyzické adresy
  - při sestavování programu musíme určit kam bude program zaveden
- **přemístitelné zavedení (relocatable loading)**
  - každý odkaz do paměti v programu obsahuje relativní adresu (adresa vztažená k určitému bodu)
  - informace o paměťových odkazech je uložena v „relocation dictionary“
  - přepočítání relativní adresy na fyzickou se provede při zavedení programu do fyzické paměti
- **dynamic run-time loading**
  - každý odkaz do paměti v programu obsahuje relativní adresu (adresa vztažená k určitému bodu)
  - program je zaveden do paměti s relativními adresami
  - relativní adresa se přepočte na fyzickou teprve při provádění instrukce

Spojování programu (linking)

- **statické spojování (static linking)**
  - vytvoří se jeden „load module“ s relativními adresami vztaženými k začátku modulu
- **dynamické spojování**
  - „load module“ obsahuje odkazy na další programy
  - **load-time dynamic linking**
    - odkazy na další programy se nahradí v okamžiku zavedení do paměti
  - **run-time dynamic linking**
    - odkazy na další programy se nahradí v okamžiku provádění instrukce

Požadavky na SP (2)

- **Ochrana**
  - každý proces musí být chráněn před nechtěnými přístupy z ostatních procesů
- **Sdílení**
  - umožnění přístupu ke společné paměti (např. několik procesů provádí stejný program)
- **Logická organizace (logický adresový prostor)**
  - lineární (jednorozměrný) adresový prostor
  - SW se obvykle skládá z modulů $\Rightarrow$ více lineárních prostorů (např. s různými právy, kompilovány v různém čase, …)
- **Fyzická organizace (fyzický adresový prostor)**
  - fyzická paměť se skládá z různých úrovní

Hierarchie pamětí

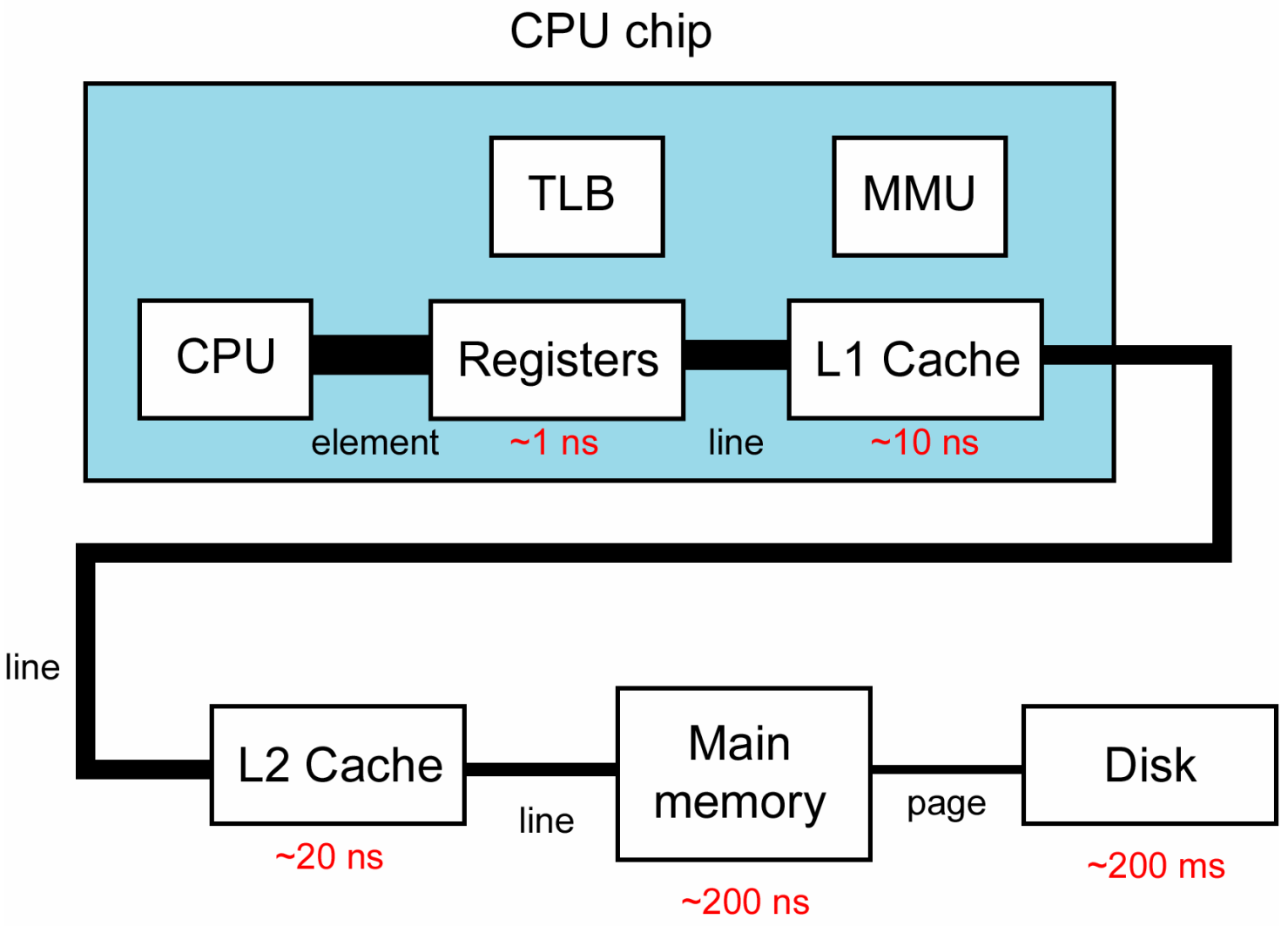
- **Inboard memory**

	velikost	čas přístupu	spravuje
registry	<1 KB	~1 ns	Compiler
L1 cache	1 MB	~10 ns	Hardware
L2 cache	0.5-8 MB	~20 ns	Hardware
hlavní paměť	16MB - 64 GB	~200 ns	Software

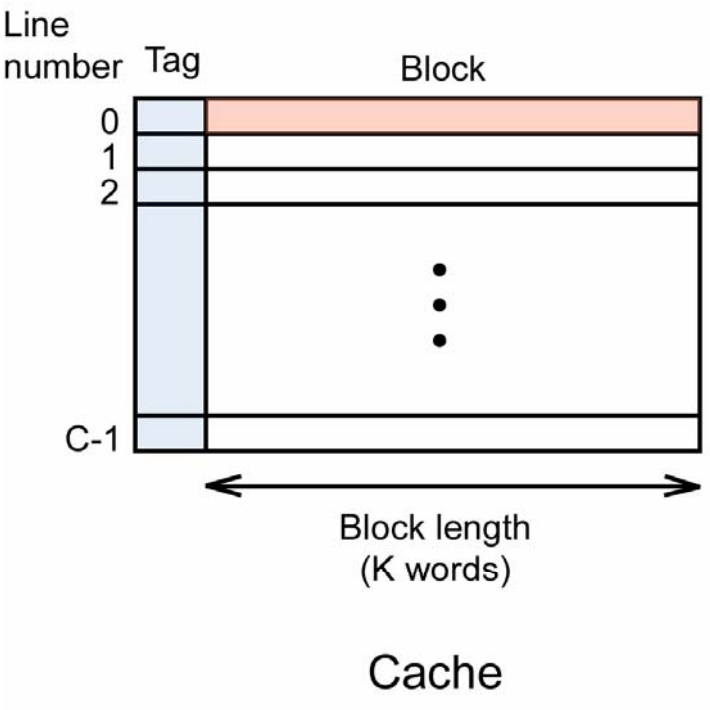
- **Outboard storage**
  - magnetické disky 1G -11TB, 200ms, Software
  - flash drive
  - CD-ROM, CD-RW
  - DVD

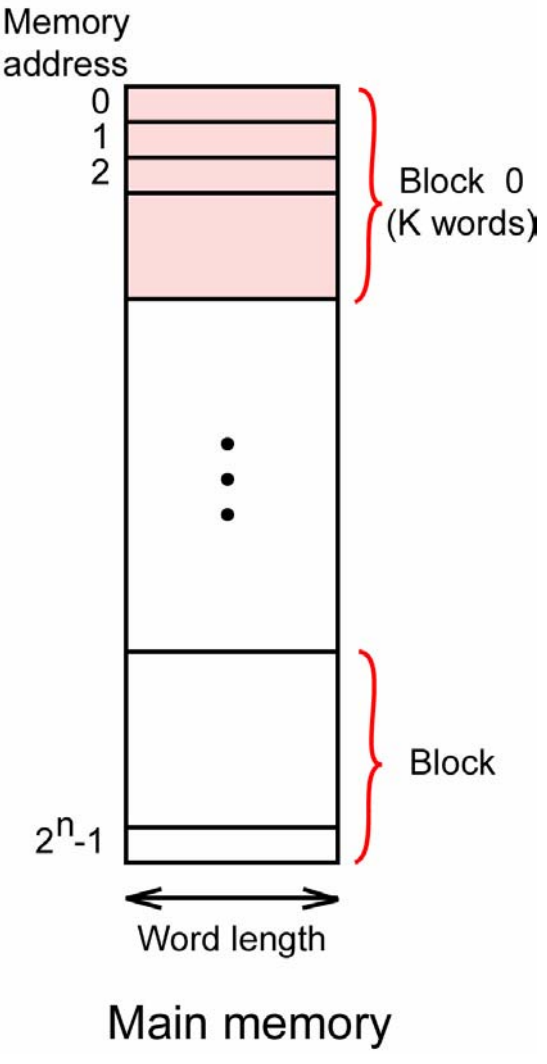
- **Off-line storage**
  - magnetické pásky 20-100GB, 100s, Software

Hierarchie pamětí (2)



Cache – Hlavní paměť





Cache design

- Velikost cache (cache size)
- Velikost bloku (block size)
- Mapovací funkce (mapping function)
- Nahrazovací algoritmus (replacement alg.)
- Strategie zapisování (write policy)

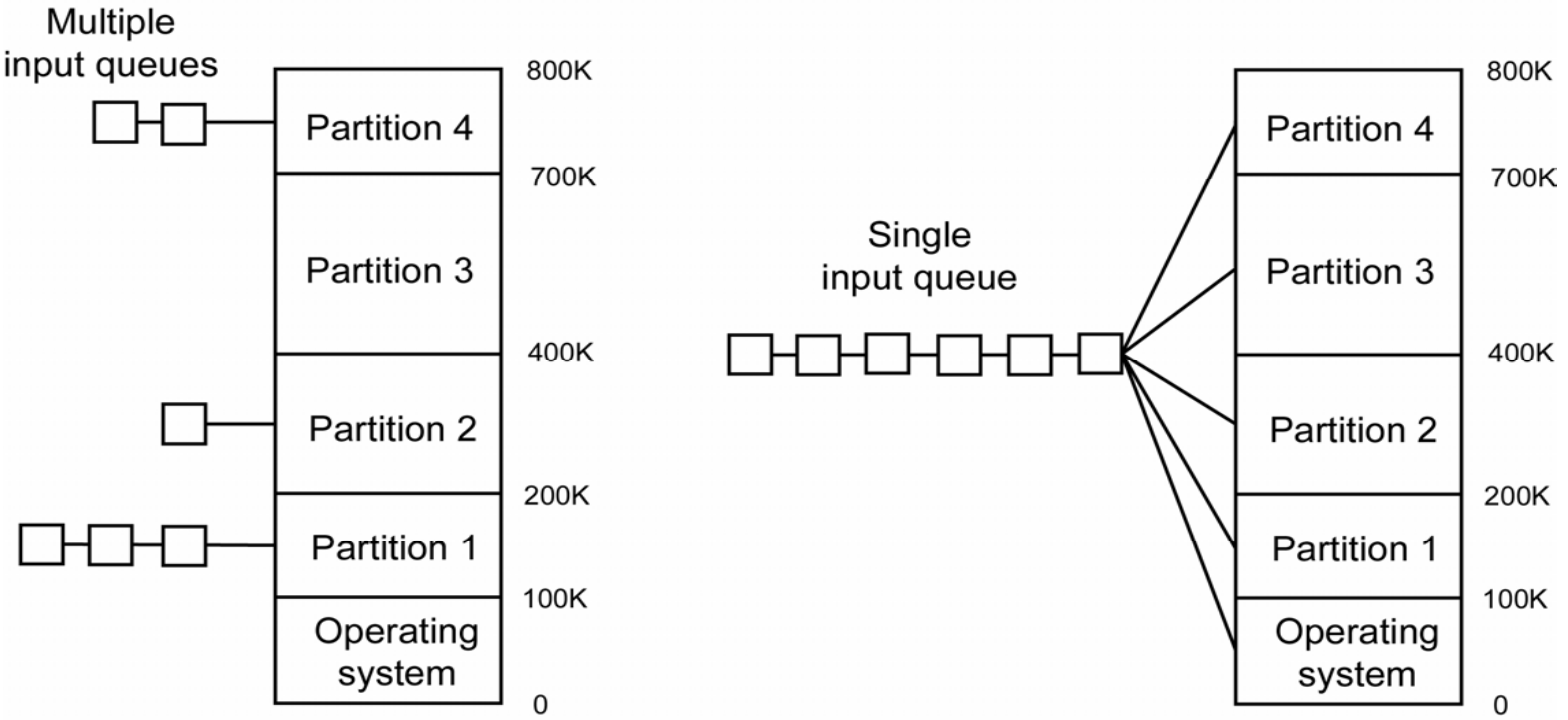
Základní techniky SP

- Fixní oblasti (fixed partitioning)
- Dynamické oblasti (dynamic partitioning)
- Jednoduché stránkování (simple paging)
- Jednoduchá segmentace (simple segmentation)

- Virtuální paměť se stránkováním (virtual memory paging)
- Virtuální paměť se segmentací (virtual memory segmentation)

Fixní oblasti

- **Paměť je rozdělena** na *n* oblastí různých velikostí (většinou při spuštění systému).
- Program je nahrán do stejně velké oblasti nebo větší.
- **Velkost paměťových oblastí se nemění** během běhu OS.



Fixní oblasti (2)

- **Oddělené vstupní fronty** pro každou oblast
  - **Nevýhoda**: nerovnoměrné obsazení front

- **Společná vstupní fronta**
  - Strategie výběru úlohy pro volnou oblast
    - ***best fit*** nalezení největší úlohy, která se vejde do oblasti
    - ***first fit*** nalezení první úlohy, která se vejde do oblasti

  - **Nevýhoda**
    - ***best fit*** znevýhodňuje malé (interaktivní) úlohy
    - ***first fit*** plýtvá místem velkých oblastí

  - **Řešení nevýhody best fit**
    - Úloha, která by mohla běžet, nesmí být předběhnuta více než *k* krát..
    - Při každém předběhnutí získá bod.
    - Pokud má úloha *k* bodů nemůže být předběhnuta.

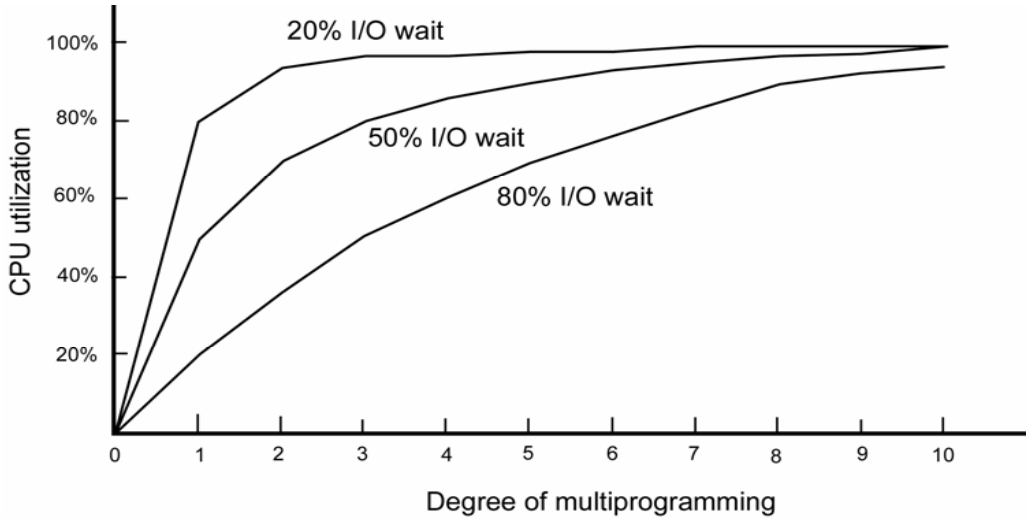
Fixní oblasti (3)

- **Výhody**
  - jednoduchá implementace
  - malé režijní náklady

- **Nevýhody**
  - **interní fragmentace** (místo uvnitř oblasti není využito na 100 %)
  - počat aktivních procesů je fixní

Modelování multiprogramování

- Multiprogramování zlepšuje využití CPU!
- **Pravděpodobnostní model využití CPU**:
  - Proces stráví zlomek svého času *p* čekáním na V/V.
  - Pokud máme současně *n* procesů v paměti, potom pravděpodobnost, že všechny procesy současně čekají na V/V je p^n.
  - **Využití CPU je** $1 - p^n$.
- Pravděpodobnostní model je **pouze přibližný odhad** (procesy nejsou na sobě nezávislé).



Přemístění a ochrana

- **Přemístění**
  - adresy proměnných, návěští skoků,… jsou **přepočítány v okamžiku nahrání** programu do paměti
  - sestavovací program musí do binárního programu zapsat informaci, která slova v programu obsahují adresy paměti, aby mohly být přepočítány

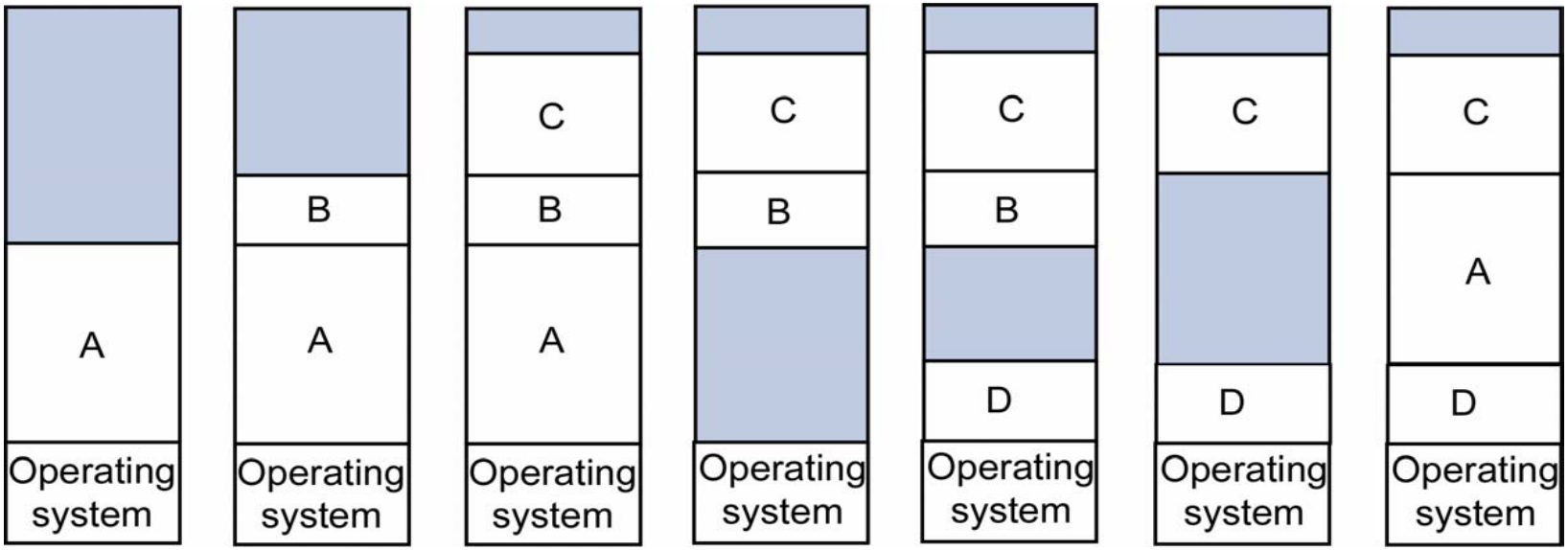
- **Ochrana**
  - paměť je rozdělena na bloky
  - každý blok je svázán s *n*-bitovým **ochranným klíčem,** který určuje zda daná úloha smí přistupovat k datům v tomto bloku (IBM 360 - 2kB bloky, 4bitový klíč).

Přemístění a ochrana (2)

- Každá oblast paměti má dva registry (**bázový** a **limitní registr**), které obsahují nejmenší a největší adresu této oblasti.
- Když přistupujeme k paměti, bázový registr je přičten k adrese paměti a výsledek je porovnán s limitním registrem.
- **Nevýhoda**: potřeba sčítání a porovnávání při každém přístupu do paměti.
- **Řešení**: speciální HW.

Dynamické oblasti

- **Počet, umístění a velikost oblastí se mění dynamicky,** tak jak jednotlivé procesy vznikají, zanikají a přesouvají se mezi hlavní pamětí a diskem.

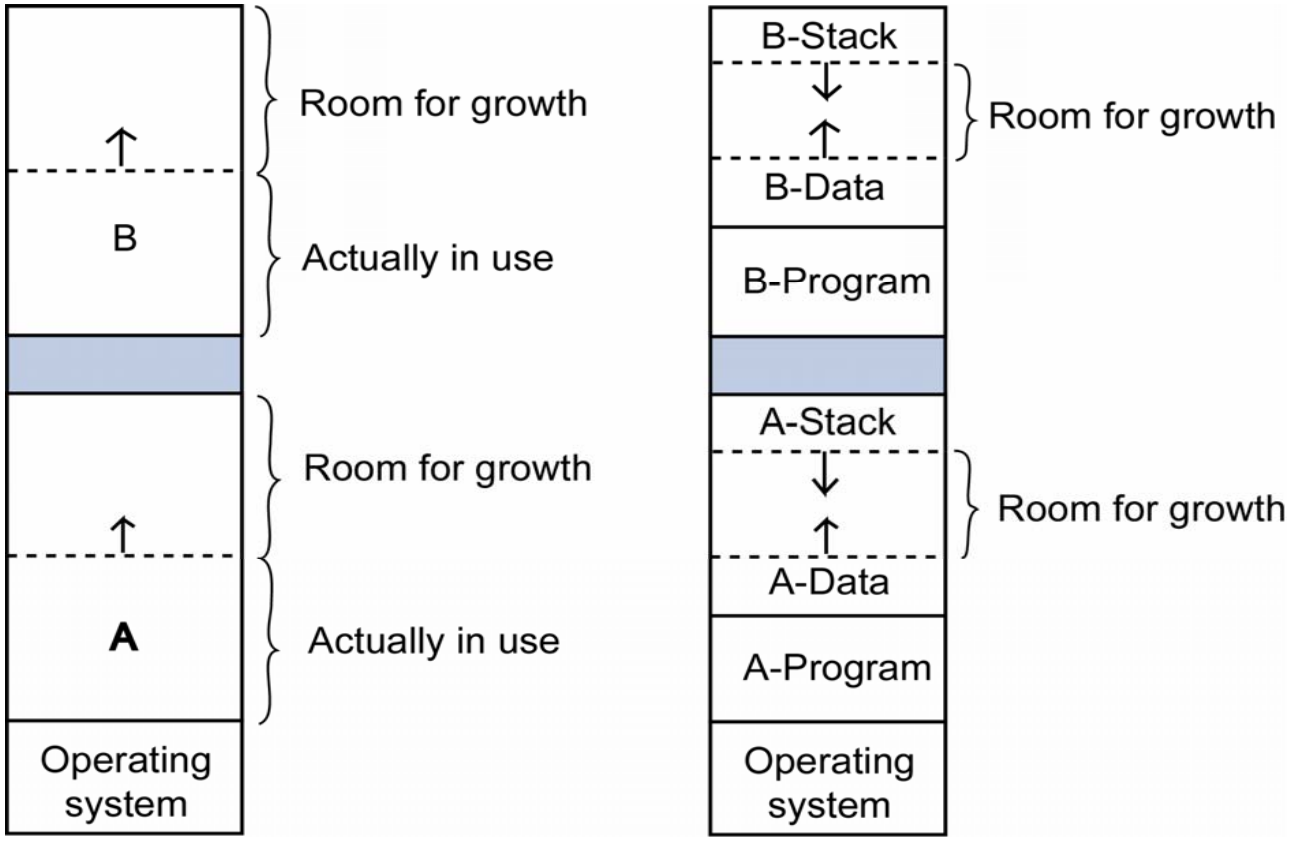


Dynamické oblasti (2)

- **Výhody**
  - „žádná“ interní fragmentace
  - efektivnější využití paměti
- **Nevýhody**
  - **externí fragmentace** (možno setřásání paměti, ale je to časově náročné)

Zvětšující se procesy

- **Datový segment** procesu **může měnit svojí velikost** během výpočtu.
- Proto musíme **alokovat více paměti** než je na počátku potřeba.



Správa použité a volné paměti

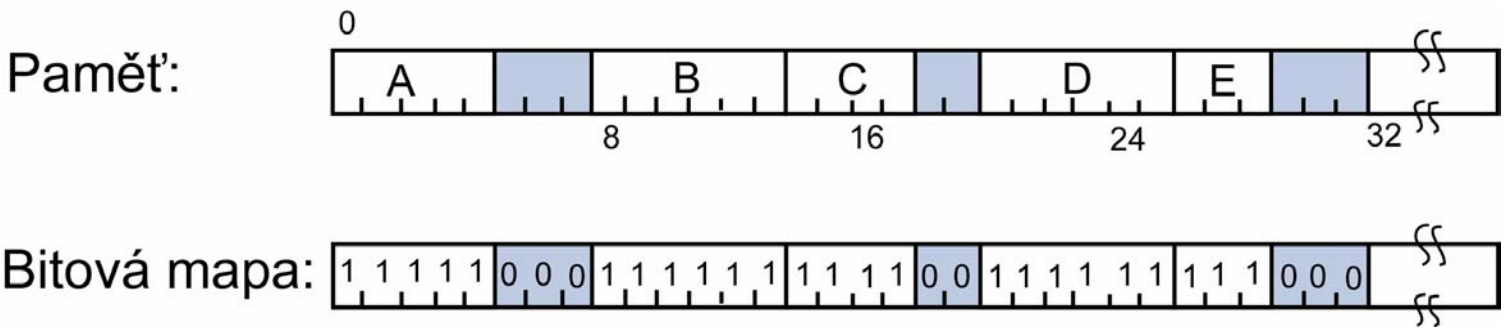
- **Jak udržovat informaci o volné a přidělené paměti?**
  - Bitové mapy.
  - Zřetězené seznamy.

Bitové mapy

- Paměť je rozdělena na **alokační jednotky** (AU, veliké několik KB).
- Každá AU má korespondující bit ve speciálním řetězci–**bitové mapě** (0-volná, 1-přidělená).

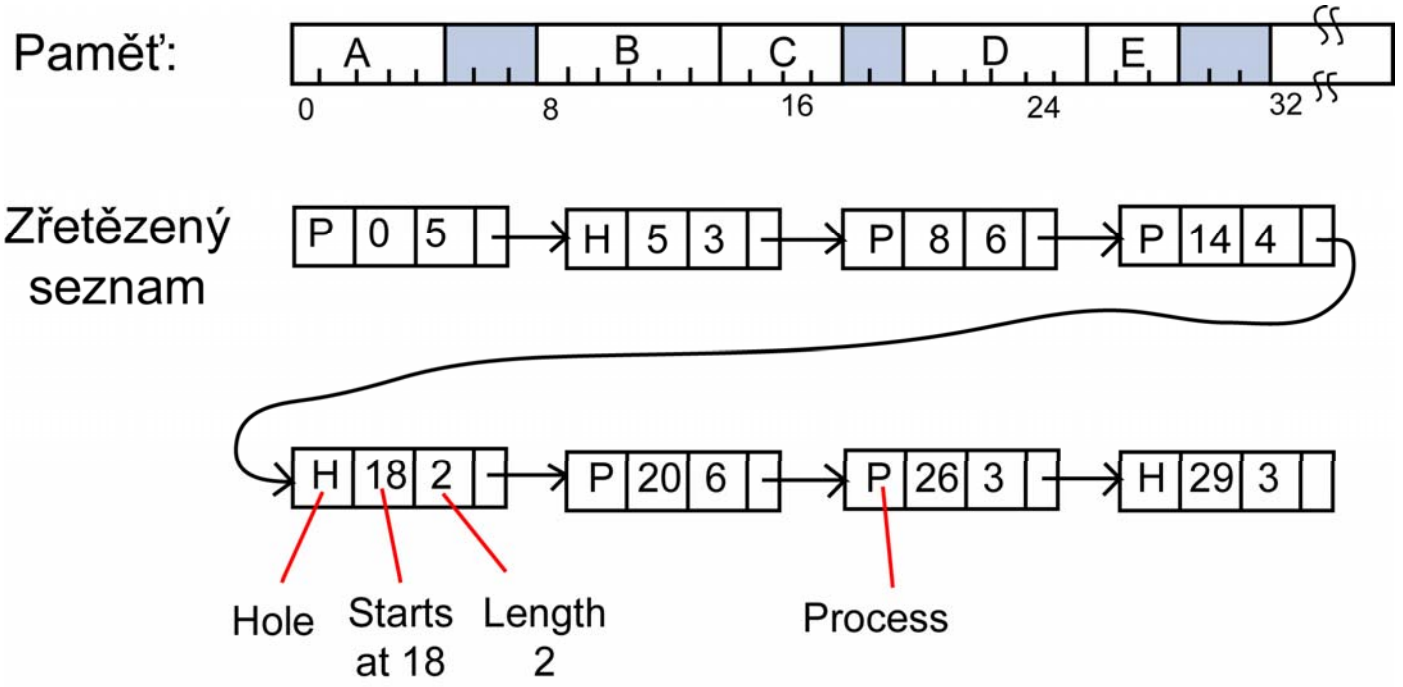
- Nalezení volných AU = nalezení **souvislého řetězce** nulových bitů.

- Problémy:
  - **Velké AU** – malá bitová mapa x plýtvání hlavní pamětí.
  - **Malé AU** – velká bitová mapa x lepší využití hlavní paměti.
  - **Hledání v bitové mapě je pomalé!**



Zřetězené seznamy

- Zřetězený seznam volných a přidělených paměťových segmentů (např. proces (P) a díra (H)).
- Seznam je **setříděn podle adres** segmentů.
- Když proces končí nebo je odložen na disk, aktualizace seznamu je jednoduchá.



Zřetězené seznamy (2)

- Paměť pro nový nebo odložený proces se může alokovat několika způsoby:
  - **first fit**
    - Nalezení první dostatečné díry od začátku seznamu.
    - Stávající díra se rozdělí na proces a díru.
  - **next fit**
    - Jako first fit, ale hledání začíná z místa, kde jsme skončili posledně.
    - Díry ze začátku seznamu nejsou preferovány.
  - **best fit**
    - Nalezení nejmenší díry, do které se daný proces vejde.
    - Stávající díra se rozdělí na proces a malou díru.
  - **worst fit**
    - Nalezení největší díry.
    - Nově zniklá díra bude dostatečné veliká pro další procesy.

Zřetězené seznamy (3)

- Na základě simulací, **next fit** vykazuje horší chování než **first fit.**
- **Best fit** je pomalejší než **first fit**, protože musíme prohledávat celý seznam.
- **Best fit** ve výsledku plýtvá pamětí více než **first fit**, protože paměť bude obsahovat velké množství malých děr.
- **Oddělené seznamy** pro procesy a díry
  - **rychlejší alokace paměti**
  - při uvolnění paměti se **sousední díry spojují**
  - vhodné použít **obousměrně zřetězený seznam**
  - **nevýhoda:** složitější a tím i pomalejší operace uvolnění paměti.

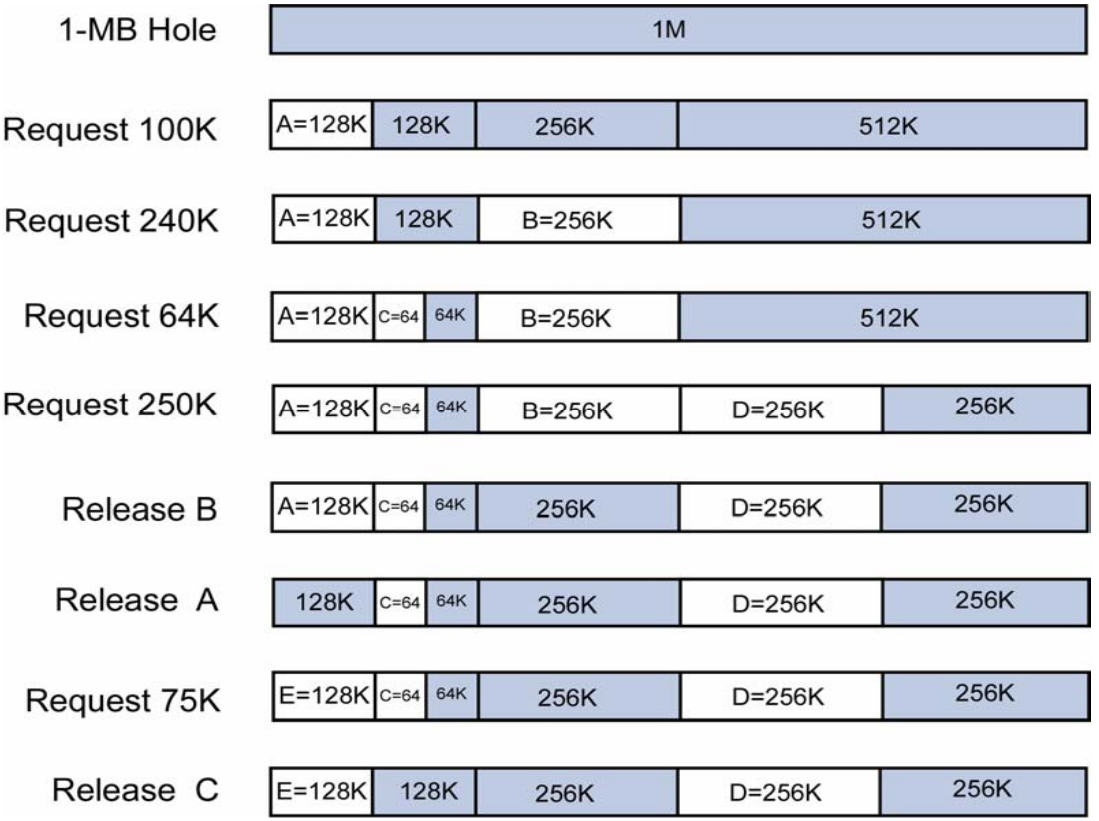
Zřetězené seznamy (4)

- **quick fit**
  - informace o dírách je udržována v několika oddělených seznamech,
  - každý seznam obsahuje informaci o dírách jejichž velikost je v určitém intervalu
  - např.:

    seznam děr od 1 kB do 5 kB
    seznam děr od 5 kB do 10 kB
    seznam děr od 10 kB do 50 kB
    ...
  - **Výhoda**
    - Rychlé nelezení volné díry.
  - **Nevýhoda**
    - Nalezení sousedních děr pro sloučení do větší díry je časově náročné.

Zřetězené seznamy (5)

- **Buddy systém = quick fit s dírami o velikosti 2^n bytů.**
- Informace o dírách je v několika oddělených seznamech pro velikosti děr 1,2,4,8, ... bytů (např. pro 1 MB, potřebujeme 21 takových seznamů.



Zřetězené seznamy (6)

- **Výhody**
  - alokace paměti je stejně rychlá jako u quick fit algoritmu,
  - slučování sousedních děr při uvolnění paměti je rychlejší než u quick fit algoritmu (nemusíme procházet všechny seznamy)

- **Použití**
  - např. pro alokaci paměti v jádře Linuxu

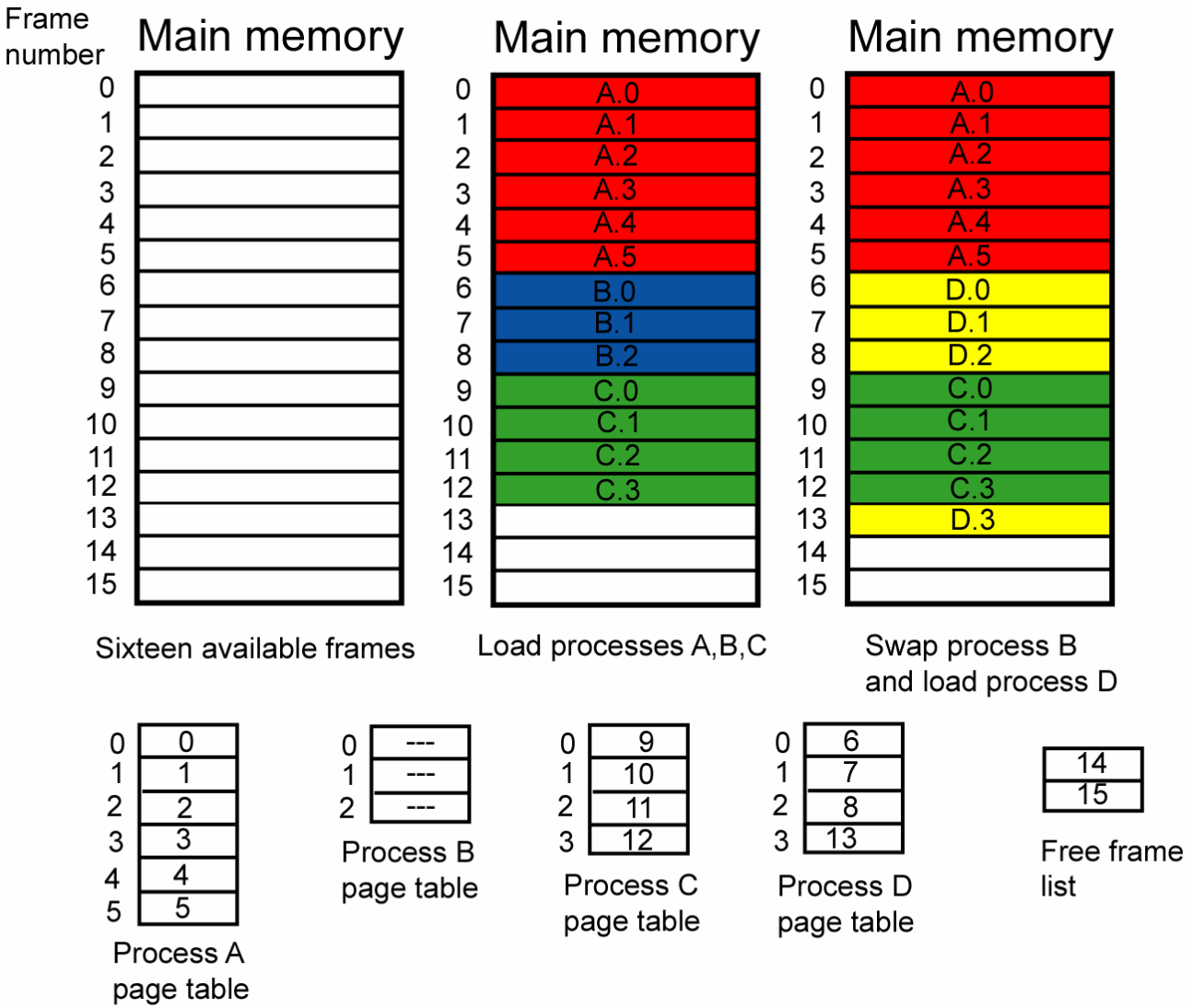
Operační systémy

Přednáška 8: Správa paměti II

Jednoduché stránkování

- **Hlavní paměť**
  - rozdělená na malé úseky stejné velikosti (např. 4kB) nazývané **rámce (frames).**
- **Program**
  - rozdělen na malé úseky stejné velikosti nazývané **stránky (pages)**
- **Velikost rámce a stránky je stejná.**
- **Celý program** je nahrán do **volných rámců** hlavní paměti.
- OS si musí pamatovat **rámce přidělené jednotlivým procesům** (např. pomocí tabulky stránek,…)
- OS si musí pamatovat **volné rámce** v hlavní paměti.

Příklad: jednoduché stránkování



Virtuální paměť

- V 32 bitovém OS (např. Unix), **jeden proces může mít 4 části**, každou o velikosti (maximum) 1GB:
  - **text** (instrukce kódu),
  - **data** (statická a dynamická),
  - **shared text** (sdílené knihovny),
  - **shared data** (sdílená paměť).

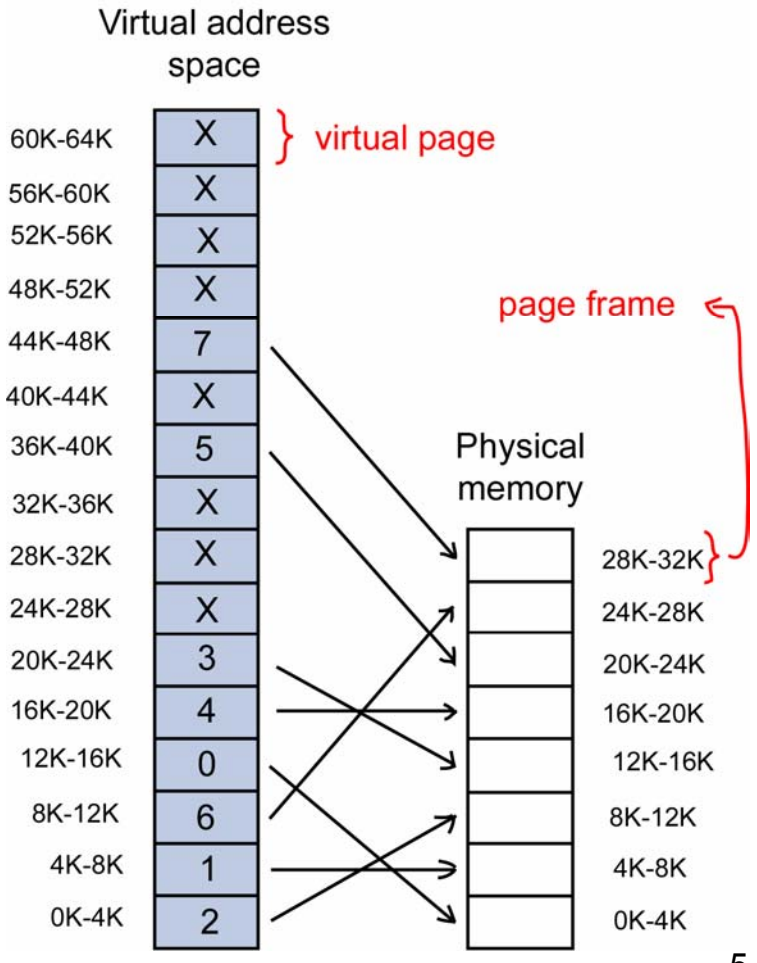
- **Problém**
  - Pokud OS umožňuje, aby bylo současně spuštěno až 64k procesů, pak bychom potřebovali dohromady 256 TB paměti.

- **Řešení**
  - **Virtuální paměť** = proces je automaticky (pomocí OS) rozdělen na menší kousky.
  - Ve fyzické paměti jsou pouze kousky aktuálně používané, zbytek procesu je na disku.

Virtuální paměť se stránkování

- Většinou je **virtuální paměť** kombinována se **stránkování**.

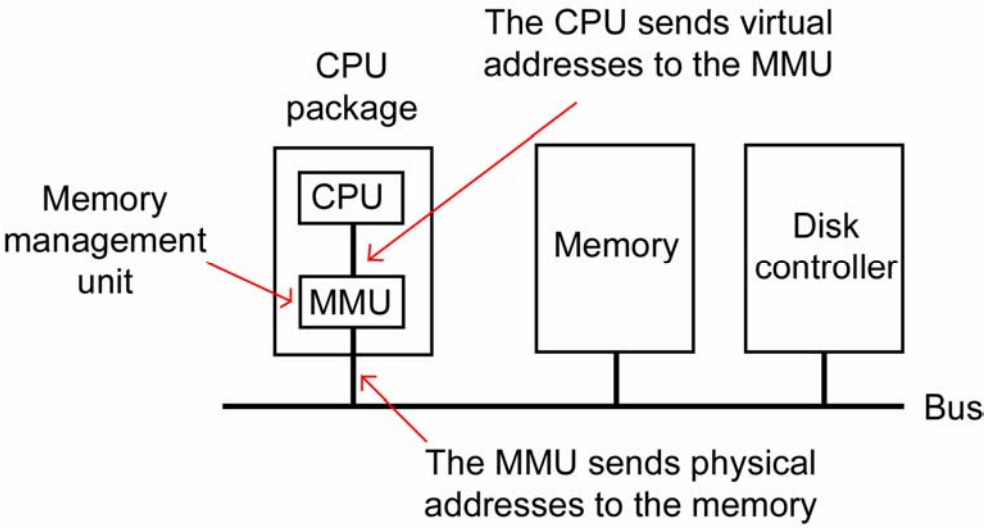
- **Princip**
  - Proces používá adresy, kterým se říká **virtuální adresy** a které tvoří **virtuální adresový prostor.**
  - Virtuální adresový prostor je rozdělen na stejně velké souvislé úseky nazývané **virtuální stránky (virtual pages)** (typicky 4KB).
  - Korespondující úseky ve **fyzické paměti** jsou nazývány **rámce stránek (page frames)**.
  - V hlavní paměti jsou pouze **stránky aktuálně používané**.



Memory Management Unit

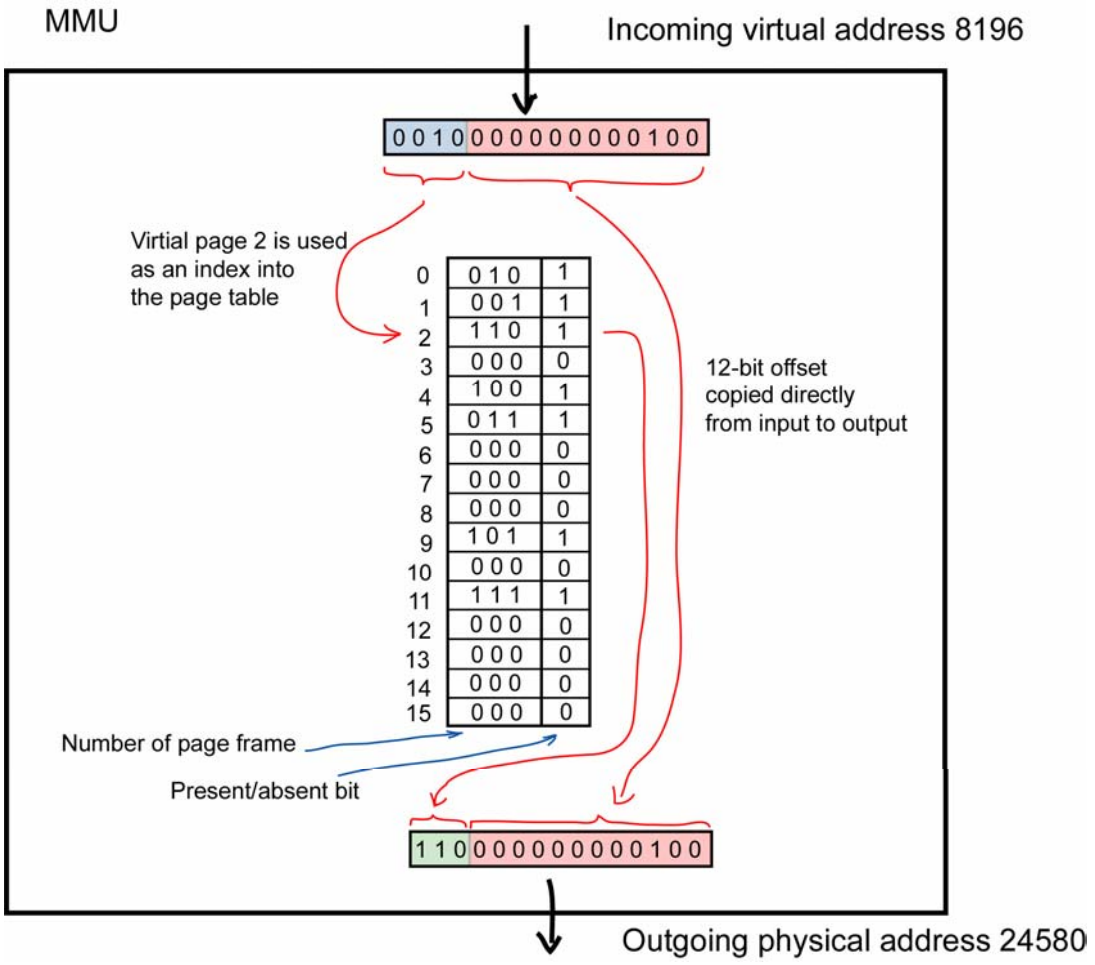
- Proces adresuje paměť pomocí virtuálních adres (např. MOV reg, va).

- **Memory Management Unit** (MMU)
  - překládá virtuální adresu na fyzickou.
- **Výpadek stránky (Page fault)**
  - Pokud není virtuální stránka ve fyzické paměti, MMU způsobí, aby CPU požádalo OS o nahrání příslušné stránky do fyzické paměti.
  - OS nejdříve definuje, který rámec fyzické paměti je třeba uvolnit, a pak do něj nahraje obsah požadované virtuální stránky z disku.



Tabulka stránek

- MMU: **číslo_fyzického_rámce = f (číslo_virtuální_stránky)**
- Zobrazení *f()* může být implementováno pomocí **tabulky stránek**.



Tabulka stránek - problémy

- Tabulka stránek může být **extrémně velká**.
  - 32-bitový virtuální adresový prostor bude mít při velikosti stránek 4-KB jeden milion stránek.
  - Tabulka stránek pak bude mít jeden milion položek.
  - Každý proces potřebuje svojí vlastní tabulku stránek (protože má svůj vlastní virtuální adresový prostor).

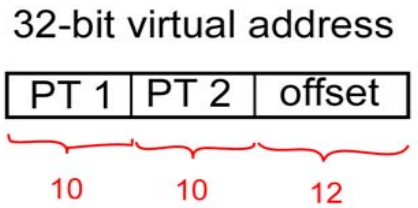
- Překlad adres by měl být **velmi rychlý.**
  - Překlad virtuální adresy na fyzickou musí být prováděn při každém přístupu do paměti.

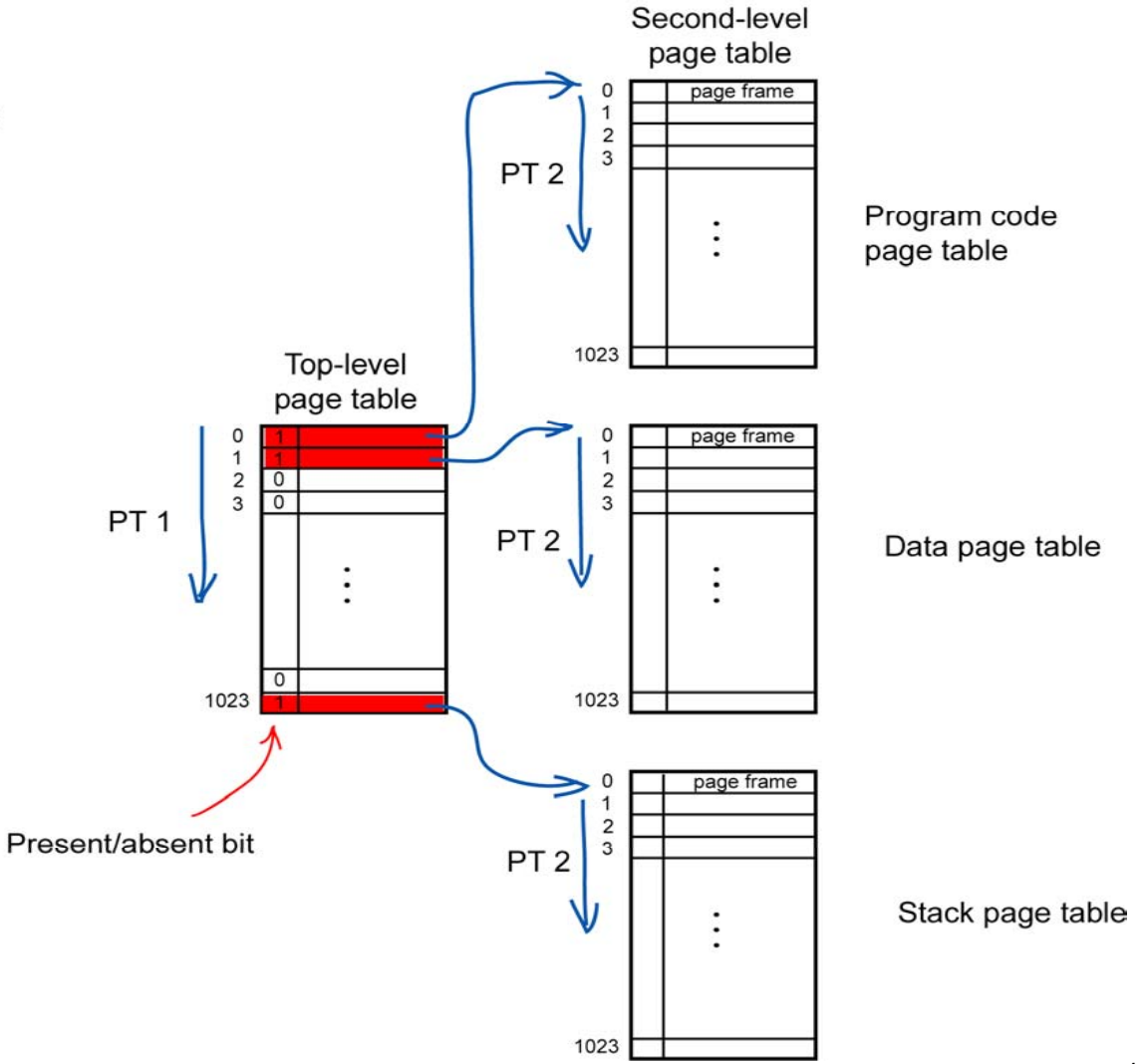
Víceúrovňová tabulka stránek

- Proces **obvykle používá pouze podmnožinu** adres svého virtuálního procesu.
- Stačilo by mít v paměti pouze ty položky z tabulky stránek, které bude OS potřebovat při překladu.
- **Příklad**:
  - Mějme **32-bitový virtuální adresový prostor** s **4KB stránkami**.
  - Předpokládejme, že **proces bude skutečně používat pouze 12MB**:
    - dolní 4MB paměti pro kód programu,
    - následující 4MB pro data,
    - horní 4MB pro zásobník.
  - Ačkoli proces má virtuální adresový prostor veliký 1MB (tzn. 1M položek v tabulce stránek), stačí mít pouze **čtyři tabulky stránek**, každou mající 1K položek:
    - top-level page table,
    - program code page table,
    - data page table,
    - stack page table.

Víceúrovňová tabulka stránek (2)

Two-level page tables



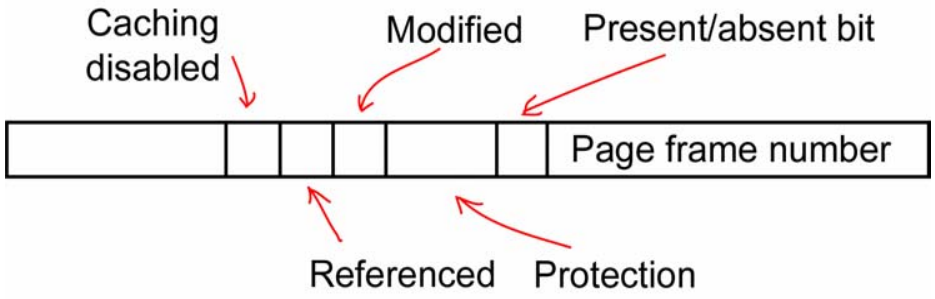


Víceúrovňová tabulka stránek (3)

- **Present/absent bity** 1021 položek v top-level page table jsou nastaveny na 0, protože virtuální stránky s nimi spojeny nebyly zatím používány.
- Při pokusu přístupu k těmto stránkám dojde k výpadku stránky a potřebné informace budou nahrány do paměti.
- Obecně lze tabulku stránek rozdělit do libovolného počtu úrovní.
- V praxi se z **důvodu rychlosti překladu adres používají pouze dvou a tříúrovňové tabulky**.
- Většina OS používá **demand paging**.
  - Když je proces spuštěn, nahrají se do RAM pouze první stránky kódu a první stránky dat.
  - Ostatní stránky budou nahrány do RAM až v okamžiku, kdy budou potřeba.
- **Výhody**: malá velikost tabulek v paměti.
- **Nevýhody**: pomalejší překlad.

Položka v tabulce stránek

- Její struktura je **závislá na architektuře CPU**, ale obvykle obsahuje:



- **Page frame number**
- **Present/absent bit**
  1 – stránka je v RAM,
  0 – stránka není v RAM,
  přístup na stránku způsobí page fault.
- **Protection bits**
  - 3 bits - reading, writing, executing.
- **Modified bit**
  - Když je obsah stránky modifikován, HW automaticky nastaví bit na 1.
  - Když OS uvolňuje rámec stránky:
    - musí obsah stránky uložit na disk pokud je „Modified bit“ roven 1.
    - jinak může nahrát do rámce rovnou novou stránku.

Položka v tabulce stránek (2)

- **Referenced bit**
  - Kdykoliv je ke stránce přistupováno (pro čtení nebo zápis), je tento bit nastaven na 1.
  - Hodnota tohoto bitu je používána algoritmy pro náhradu stránek.

- **Caching disabled bit**
  - Je důležitý pro stránky, které jsou mapovány na registry periferních zařízení.
  - Pokud čekáme na V/V (např. v cyklu), musíme použít hodnoty z fyzických HW registrů, nikoliv (starý) obsah v paměti.

Translation Lookaside Buffer (TLB)

- Většina programů provádí velký počet přístupů k malému počtu stránek.
- **Translation Lookaside Buffer** (TLB)
  - Je organizovaný jako **asociativní paměť.**
  - Obsahuje **posledně používané položky tabulek stránek.**
  - TLB je obvykle uvnitř MMU a obsahuje desítky položek.

Valid	Virtual page	Modified	Protection	Page frame
1	140	0	RWX	31
1	12	0	R	12
1	25	1	R X	13
1	256	0	RWX	23
1	2	1	RWX	5
1	311	1	RWX	78

Translation Lookaside Buffer (2)

- Při překladu virtuální adresy(VA), MMU nejdříve **hledá informaci o VA v TLB**.
- Hledávání v TLB probíhá **paralelně.**

- Pokud informace o VA **existuje v TLB**, MMU použije tuto informaci pro překlad VA a nemusí hledat v tabulce stránek.

- Pokud informace **v TLB není**, MMU vyvolá **TLB fault**.
  OS pak musí **načíst informaci z tabulky stránek.**

Invertovaná tabulka stránek

- V klasické tabulce stránek **číslo virtuální stránky slouží jako index do tabulky**.

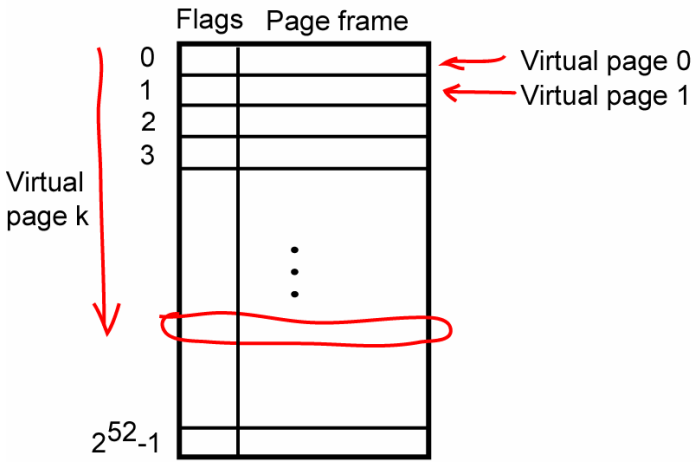
- V **32 bitových počítačích**, každý proces má 32 bitovou virtuální adresu. Při velikosti stránky 4kB, tabulka stránek každého procesu má 1M položek. Se 4B na každou položku, **tabulka stránek zabírá 4MB**.

- V **64 bitových počítačích** se 64 bitovou virtuální adresou je situace ještě více zřejmější. Při 4kB stránkách, tabulka stránek má 2^{52} položek.

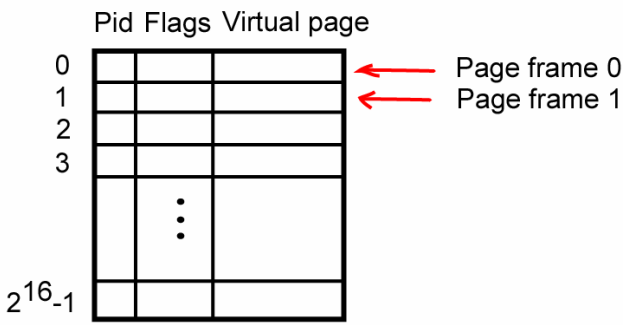
Invertovaná tabulka stránek (2)

- Ačkoliv virtuální adresový prostor je obrovský, fyzický prostor RAM je stále malý.
- Tabulka stránek muže být organizována kolem **fyzické paměti**.
- V **invertované tabulce stránek**, *i*-th položka obsahuje informaci o virtuální stránce, která je nahrána v **rámci *i***.

Traditional page table with an entry for each of the 2^{52} pages



256-MB physical memory has 2^{16} 4-KB page frames



Invertovaná tabulka stránek (3)

- Invertovaná tabulka stránek s obvykle **používá společně s TLB**.
  - Při nalezení v TLB, se invertovaná tabulka nepoužije.
  - Jinak musíme hledat v invertované tabulce stránek.

- Sekvenční hledání v tabulce může být urychleno pomocí **rozptylovací funkce**.

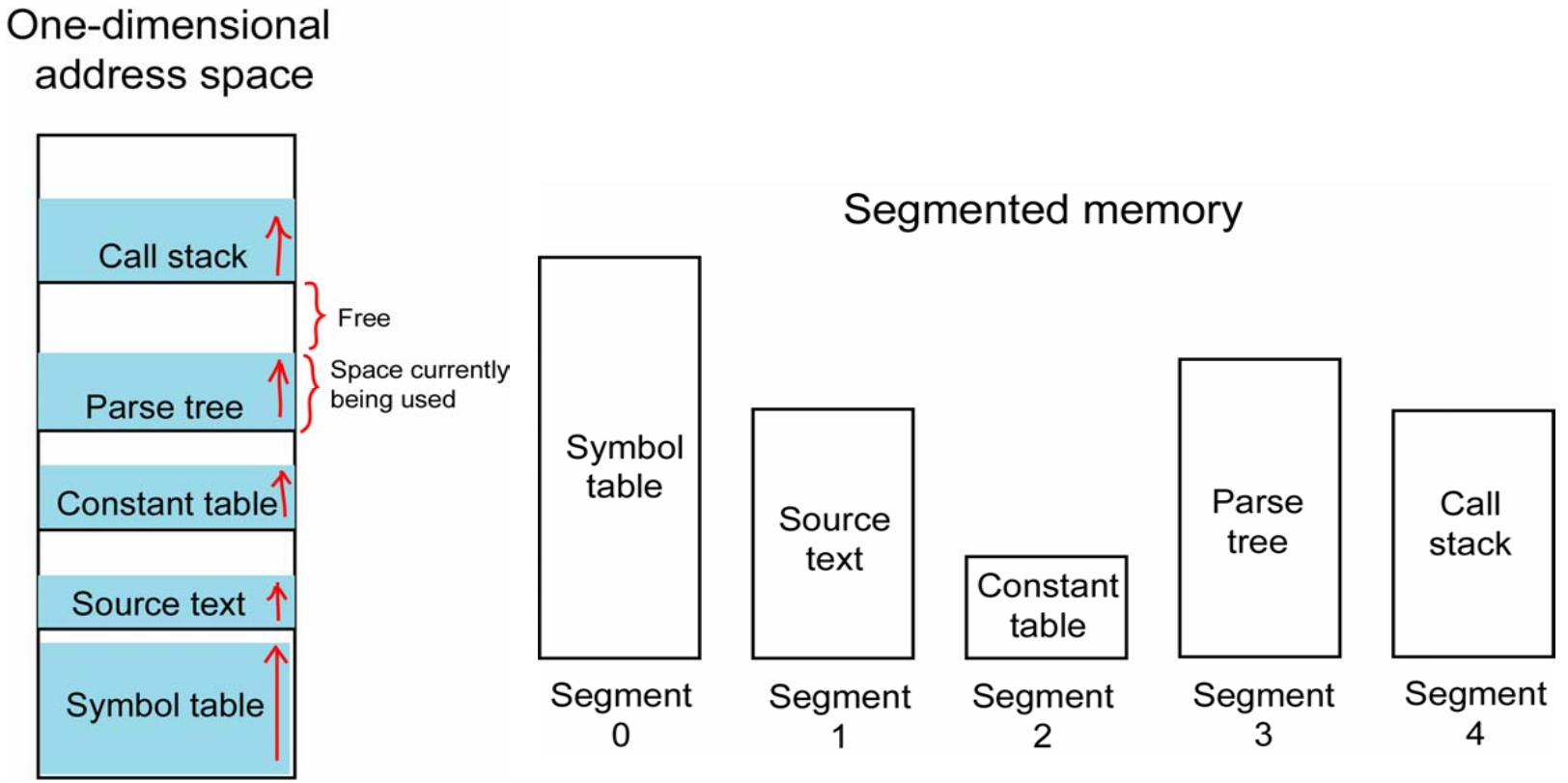
Virtuální paměť vs. Segmentace

- **Virtuální paměť**
  - Proces má jednorozměrný virtuální adresový prostor.
  - Pro některé problémy, **dva nebo více oddělených adresových prostorů** (segmentů) je vhodnější.

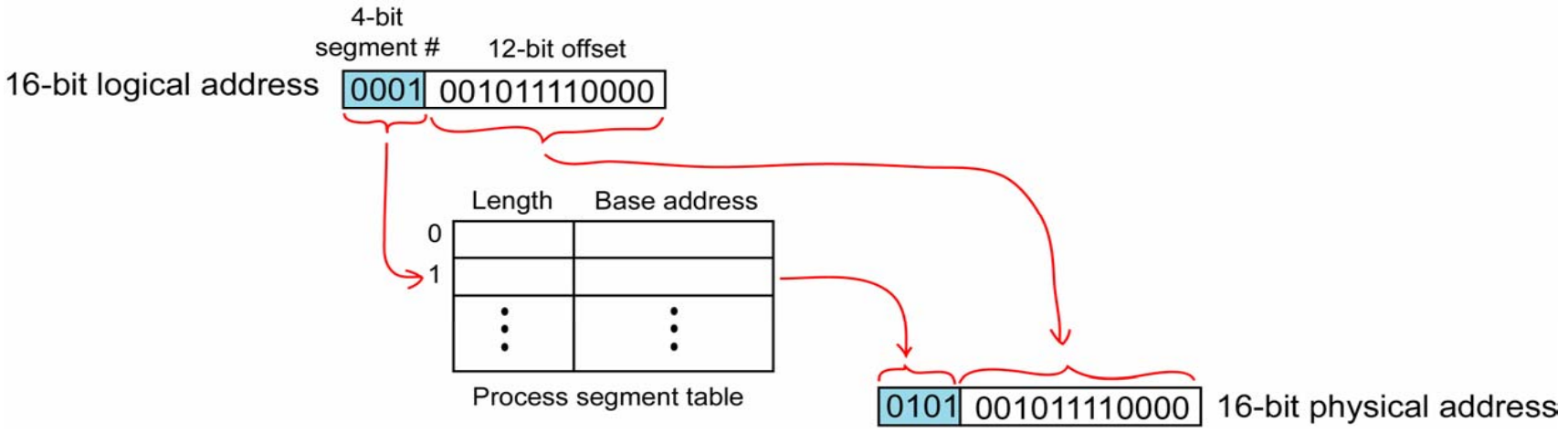
- **Segmentace**
  - Virtuální adresový prostor procesu je rozdělen na několik **segmentů**.
  - Segment je lineární posloupnost adres, od 0 do nějaké maximální adresy.
  - Různé segmenty mohou mít různé délky, **délka segmentu se může měnit během výpočtu.**
  - Různé segmenty mohou mít **rozdílnou ochranu** a mohou být **sdílené**.

Jednoduchá segmentace

- **Příklad:** Překladač si udržuje několik tabulek a datových struktur, jejichž velikost se během překladu dynamicky mění.



Jednoduchá segmentace



- **Logická adresa** se skládá ze dvou částí: **číslo segmentu** a **offsetu**.
- Segmentace je obvykle **viditelná pro programátora**.

Segmentace se stránkováním

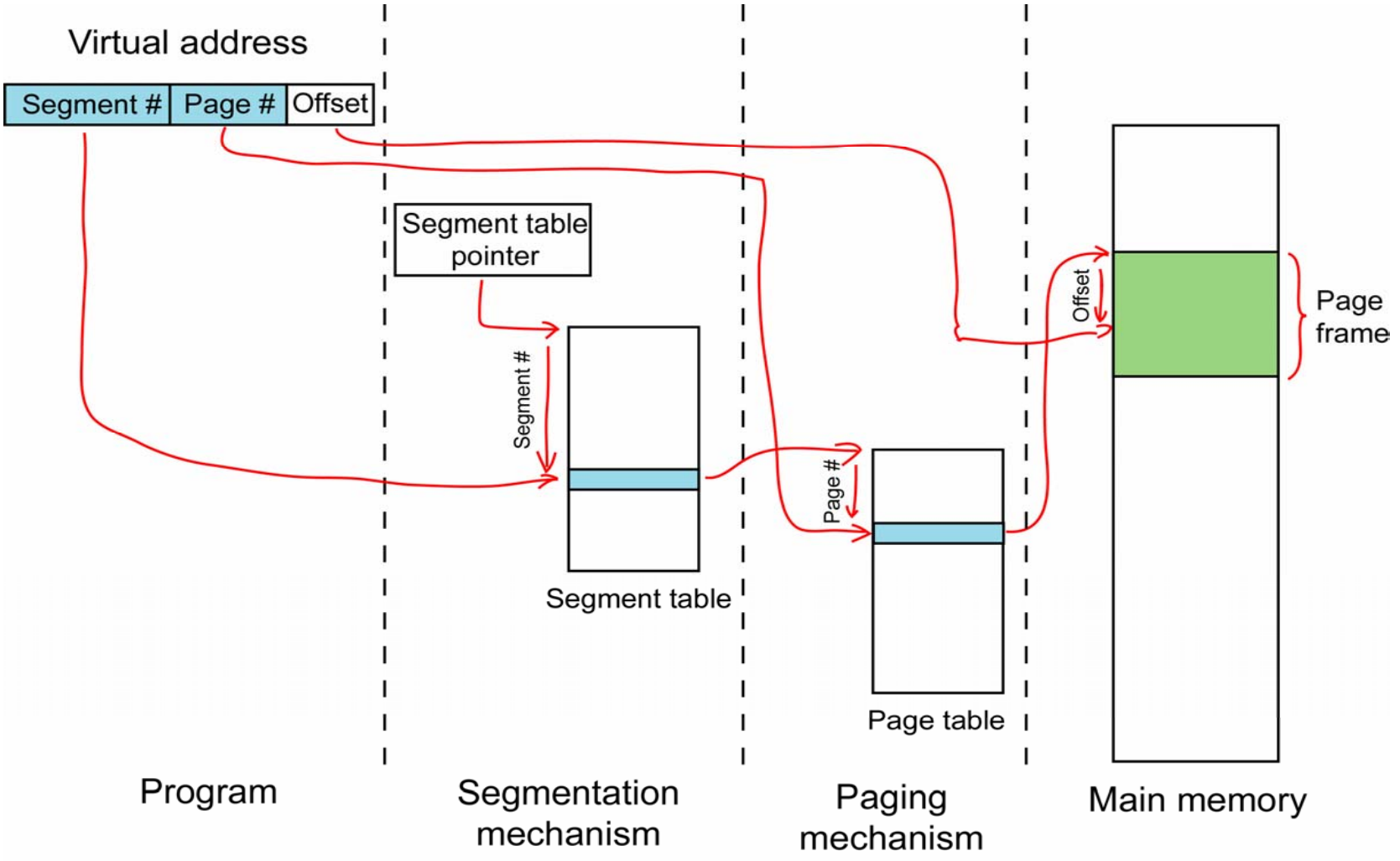
- **Stránkování**
  - Je transparentní pro programátora.
  - Eliminuje externí fragmentaci a poskytuje efektivní využití hlavní paměti.
- **Segmentace**
  - Je viditelná pro programátora.
  - Je vhodná pro dynamicky rostoucí datové struktury, modularitu, a podporuje sdílení a ochranu.
- **Segmentace se stránkováním**
  - Virtuální adresový prostor je rozdělen na **několik segmentů**.
  - Každý segment se skládá z **stejně velkých stránek**, které jsou stejně velké jak rámce v hlavní paměti.

Segmentace se stránkováním (2)

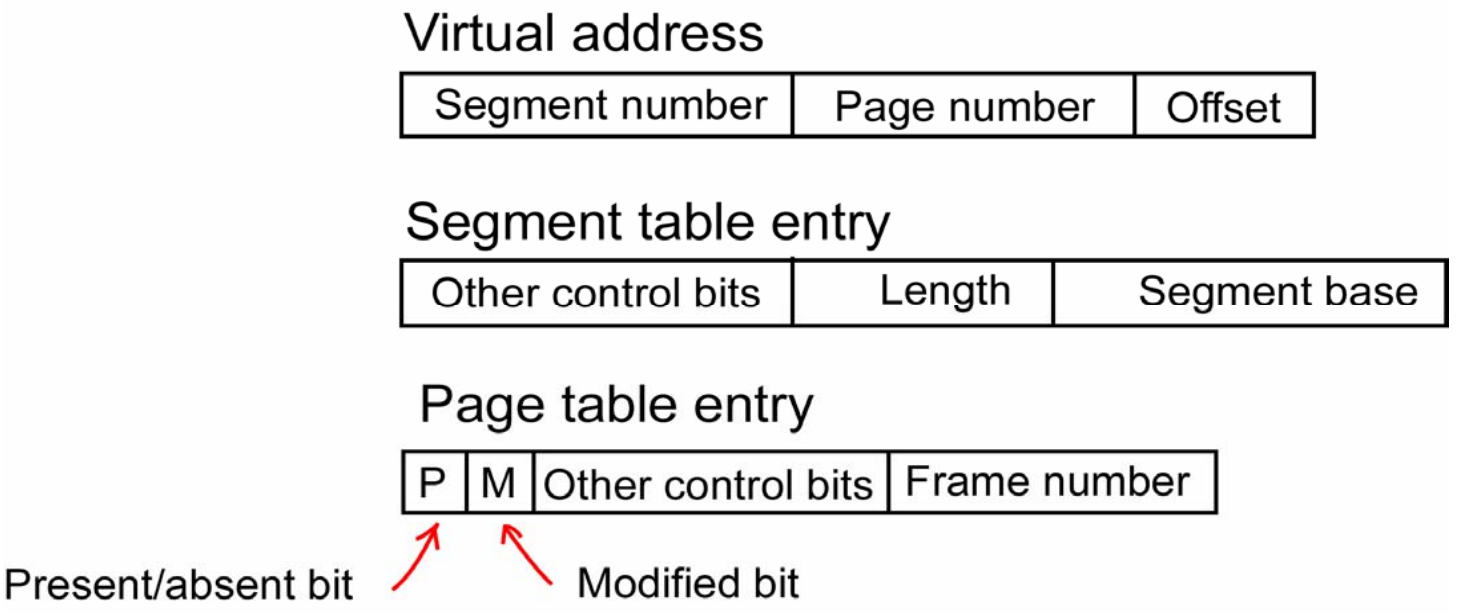
- Z hlediska **programátora**
  - Virtuální adresa se skládá z **čísla segmentu** a **offsetu uvnitř segmentu**.

- Z hlediska **systému**
  - Offset segmentu se skládá z **čísla stránky** a **offsetu uvnitř stránky**.

Segmentace se stránkováním (3)

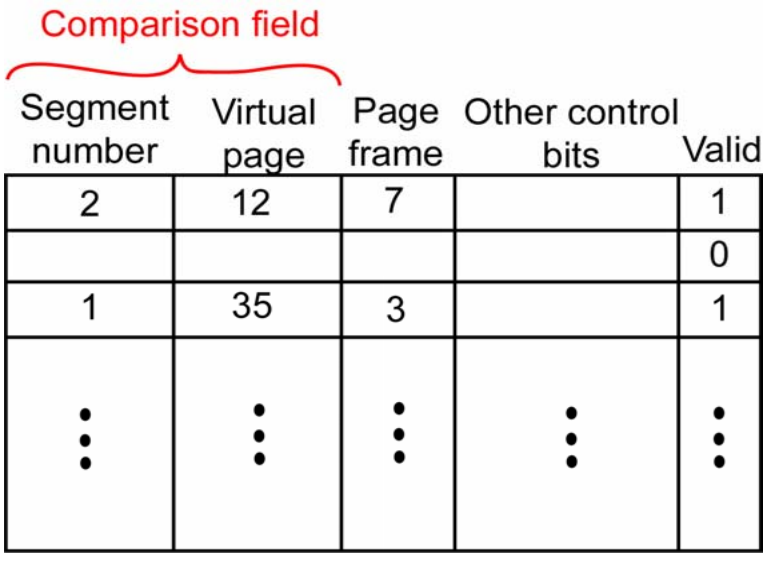


Segmentace se stránkováním (4)



- „Segment base“ ukazuje na začátek tabulky stránek pro daný segment.
- „Other control bits“ v tabulce segmentů slouží pro definici přístupových práv a sdílení mezi procesy.

Segmentace se stránkováním (5)



- Segmentace se stránkováním se může **používat společně s TLB**.

- Při překladu virtuální adresy:
  - MMU se nejdříve podívá zda není informace v TLB.
  - Pokud ano, použije pro překlad číslo rámce z TLB.
  - Jinak MMU hledá v tabulce segmentů,…

Operační systémy

Přednáška 9: Správa paměti III

Strategie nahrání (Fetch policy)

- Určuje, kdy má být virtuální stránka nahrána do hlavní paměti.

- **Stránkování na žádost (demand paging)**
  - Virtuální stránky jsou nahrávány do hlavní paměti až v okamžiku, když se na ně proces pokusí přistoupit.

- **Předstránkování (prepaging)**
  - Virtuální stránky se nahrají do hlavní paměti ještě dříve než se na ně přistupuje.

Strategie umístění (Placement policy)

- Určuje, kam do hlavní paměti bude stránka nahrána.
- V systémech s čistou segmentací hraje **důležitou roli** (např. best-fit, first-fit,...).
- V systémech se stránkováním je **bezpředmětná**.

Strategie nahrazování (Replacement policy)

- Určuje, které stránky v hlavní paměti budou nahrazeny.

- **Rezidentní velikost (Resident set size)**
  - Kolik rámců stránek v hlavní paměti bude alokováno pro každý proces.
- **Rezidentní rozsah (Resident scope)**
  - Určuje zda kandidáti pro náhradu stránky budou pouze stránky patřící procesu, který způsobil výpadek stránky, a nebo všechny stránky (i stránky patřící ostatním procesům) v hlavní paměti.
- **Nahrazovací algoritmy**
  - Z množiny kandidátů vybere stránku, která se nahradí (např. Aging alg., WSclock alg.,...) .

Strategie nahrazování (2)

- **Rezidentní velikost**
  - **fixed-allocation** policy
    - Každý proces dostane přidělen fixní počet rámců v hlavní paměti.
    - Tento počet může záviset na typu procesu (interaktivní,, dávková,…), může být určen programátorem, administrátorem,...
    - Nevýhoda:
      - Pokud je přiděleno málo rámců, bude docházet k častým výpadkům stránek.
      - Pokud je přiděleno moc rámců, do hlavní paměti se vejde pouze malý počet procesů.
  - **variable-allocation** policy
    - Počet rámců přidělených procesu se dynamicky mění podle potřeby.
    - Velký počet výpadků stránek obvykle indikuje nutnost přidat další rámce danému procesu (PFF – Page Fault Frequency alg.).
  - Variable-allocation policy je efektivnější, ale vyžaduje softwarové řešení v jádru OS.

Strategie nahrazování (3)

- **Rezidentní rozsah**
  - A **local** replacement policy
    - Mezi kandidáty pro náhradu stránek patří pouze stránky patřící procesu, který způsobil výpadek stránky.
  - A **global** replacement policy
    - Mezi kandidáty patří všechny stránky v hlavní paměti.
  - Pro nahrazování stránek existují tyto možnosti
    - **Fixed-allocation, local scope**
    - **Variable-allocation, global scope**.
    - **Variable-allocation, local scope**

Strategie ukládání (Cleaning policy)

- Určuje, kdy se modifikované stránky budou zapisovat na disk.
- **Ukládání na žádost (Demand cleaning)**
  - Stránka je uložena na disk až v okamžiku nahrazování.

- **Průběžné ukládání (Precleaning policy)**
  - Modifikované stránky jsou průběžně ukládány na disk. Ukládání je efektivnější protože probíhá ve větších objemech.

- **Page buffering**
  - Ukládáme pouze stránky, které by mohly být nahrazeny (spojení ukládání a nahrazování do jednoho kroku).
  - Stránky vhodné pro náhradu jsou umístěny do dvou seznamů: „modifikované“ a „nemodifikované“.
  - Stránky ze seznamu „modifikované“ jsou periodicky v dávce zapisovány na disk a přesunuty do seznamu „nemodifikované“.
  - Stránky ze seznamu „nemodifikované“ jsou buď ze seznamu vyloučeny, pokud se k nim bude ještě přistupovat, a nebo přepsány pokud jsou nahrazeny.

Řízení zatížení (Load control)

- Pokud proces způsobuje časté výpadky stránek, je označen jako **„přetížený“ (thrashing)**.
- Pokud **pracovní množiny (working sets) všech procesů překročí kapacitu** hlavní paměti, dojde k **přetížení celého systému**.
- Způsob jak snížit přetížení hlavní paměti je **odložení (swap)** některých procesů na disk a uvolněné rámce přidělit ostatním procesům.
- Periodicky, některé procesy přesunuty z paměti na disk a jiné z disku zpět do paměti.

Alokace odkládacího prostoru (swap area)

- **Static swap area**
  - Když je proces spuštěn, je pro něj rezervován alokační prostor.
  - **Výhoda:**
    - Procesy jsou odkládány na stejné místo na disku.
  - **Nevýhoda:**
    - Alokujeme více než je v daném okamžiku nutné.
    - Problém se zvětšujícími se procesy.

- **Dynamic swap area**
  - Alokujeme místo na disku pro každou stránku až při odkládání a uvolníme ho pří zpětném přesunu do paměti.
  - **Nevýhoda:**
    - Proces muže být ukončen při nedostatku odkládacího prostoru.
    - Složitější SW pro správu odkládacího prostoru.

Algoritmy pro náhradu stránek

- **Všechny rámce** stránek v hlavní paměti jsou **obsazené**.
- Potřebujeme **nahrát další virtuální stránku** do hlavní paměti.
- Algoritmus pro náhradu stránek musí **určit stránku, která bude nahrazena**.
- Většina algoritmů je založena na **principu lokality** (stránky, ke kterým se přistupovalo v nedávné minulosti, se bude přistupovat s velkou pravděpodobností v blízké budoucnosti).

Optimální algoritmus

- **Princip:**
  - **Nahradí se stránka**, která již **nebude používána** (pokud taková existuje).
  - Jinak nahradíme stránku, která bude **používána za nejdelší dobu** od tohoto okamžiku.

- **Realizace**:
  - Každé stránce přiřadíme číslo, které říká, kolik instrukcí se musí provést před tím, než se k této stránce bude přistupovat.
  - Optimální algoritmus nahradí stránku s nejvyšším číslem.

- **Výhody:**
  - Dosahuje nejlepších výsledků.

- **Nevýhody:**
  - Musíme znát budoucnost.
  - V praxi nelze použít.
  - Slouží pro porovnávání ostatních algoritmů.

Not Recently Used Alg. (NRU)

- Každé stránce jsou přiřazeny dva bity *R* a *M*:
  R bit je nastaven když ke stránce **přistupujeme** (čtení/zápis),
  M bit je nastaven, když stránku **modifikujeme** (zápis).

- Tyto bity jsou uloženy např. v tabulce stránek.
- Jsou **aktualizovány hardwarem** při každém přístupu na stránku.

- **Při spuštění procesu** jsou oba bity **vynulovány**.

- **Periodicky** (např. při přerušení od časovače) **je bit R nulován** tak, abychom rozlišili, které stránky byly používány nedávno.

NRU (2)

- Při **výpadku stránky**, lze stránky rozdělit do 4 kategorií:

  Kategorie 0: R=0 M=0,

  Kategorie 1: R=0 M=1,

  Kategorie 2: R=1 M=0,

  Kategorie 3: R=1 M=1.

- Algoritmus NRU náhodně **nahradí stránku z nejnižší kategorie, která není prázdná**.

- NRU je jednoduchý na pochopení a implementaci, a poskytuje odpovídající výsledky.

First-In, First-Out Alg. (FIFO)

- OS si udržuje **seznam všech stránek**, které jsou právě v hlavní paměti.
  - **Nejstarší stránky** jsou **na začátku** seznamu.
  - **Posledně nahrané stránky** jsou **na konci** seznamu.

- Při výpadku stránky, je nejstarší **stránka ze začátku seznamu vyhozena** z hlavní paměti a **nová stránka je přidána na konec seznamu.**

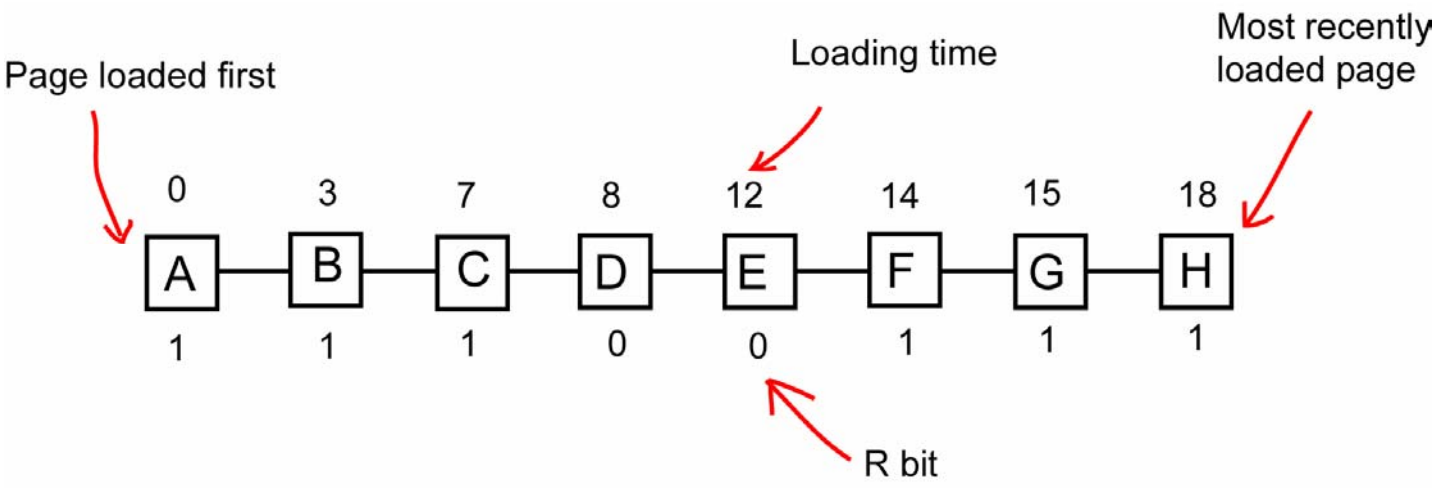
- **Nevýhoda:**
  - FIFO bere v úvahu pouze, kdy se stránka načetla do hl. paměti, ale nikoliv jak často se ke stránce přistupuje.

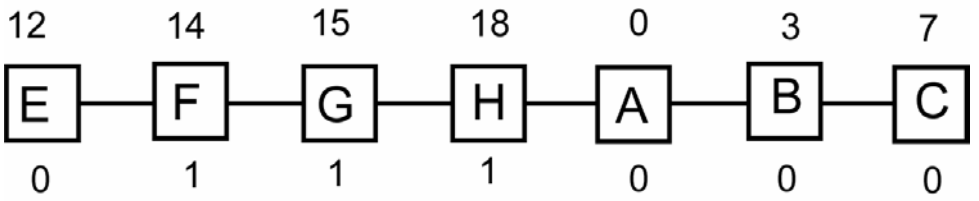
Second Chance Alg.

- Jednoduchá modifikace FIFO pomocí **bitu R,** který se nastavuje při přístupu ke stránce a periodicky se nuluje.

- **Princip:**
  - Při výpadku stránky **procházíme seznam od začátku**.
  - Pokud ***R* bit je 0**, je stránka stará a nepoužívaná, proto ji **nahradíme**.
  - Pokud ***R* bit je 1**, potom tento **bit vynulujeme, stránku přesuneme na konec seznamu a pokračujeme v hledání**.

Second Chance Alg. (2)



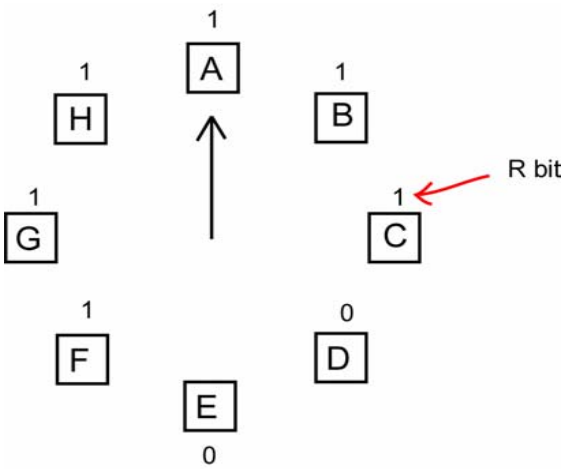


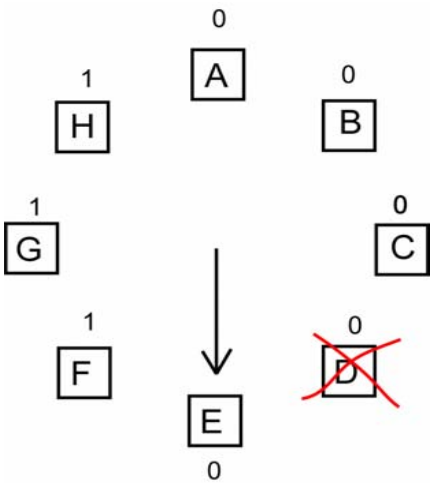
The Clock Alg.

- Modifikace algoritmu „second chance“.
- **Informace o virtuálních stránkách** je udržována v **cyklické frontě** orientované ve směru hodinových ručiček.
- Při výpadku stránky, začneme hledat od stránky, na kterou ukazuje ručička:

  Pokud ***R* = 0**, potom stránku nahradíme,

  Pokud ***R* = 1**, potom vynulujeme bit ***R*** a posuneme ručičku a hledáme dál.





Least Recently Used Alg. (LRU)

- Dobrá aproximace optimálního algoritmu.

- **Princip:**
  - **Stránky hodně používané** během několika posledních instrukcí **budou s velkou pravděpodobností ještě používány** během několika následujících instrukcí.

- **LRU**
  - Při výpadku stránky **nahradíme stránku, která nebyla používána po nejdelší dobu**.

- **Teoreticky realizovatelné, ale drahé.**
  - Úplná implementace LRU
    - seznam stránek setříděný podle doby přístupu,
    - aktualizován při každém přístupu do paměti.

LRU - HW implementace I

- Máme **speciální 64-bit HW čítač *C***.
- **Každá položka v tabulce stránek má políčko**, které obsahuje kopii čítače ***C***.

- Čítač ***C*** je automaticky inkrementován při každé instrukci.

- Při přístupu na danou stránku
  - aktuální **hodnota čítače C je uložena do položky v tabulce stránek** patřící dané stránce.

- Při výpadku stránky, **nahradíme stránku s nejmenší hodnotou C**.

LRU - HW implementace II

- Pro počítač s ***n*** rámci stránek, LRU hardware udržuje **matici *n* x *n* bitů**, na začátku inicializovaných na 0.

- Při přístupu ke stránce uložené v rámci ***k*** :
  - nejdříve HW nastaví všechny bity v **řádku *k* na 1**,
  - potom nastaví všechny bity ve **sloupci *k* na 0**.

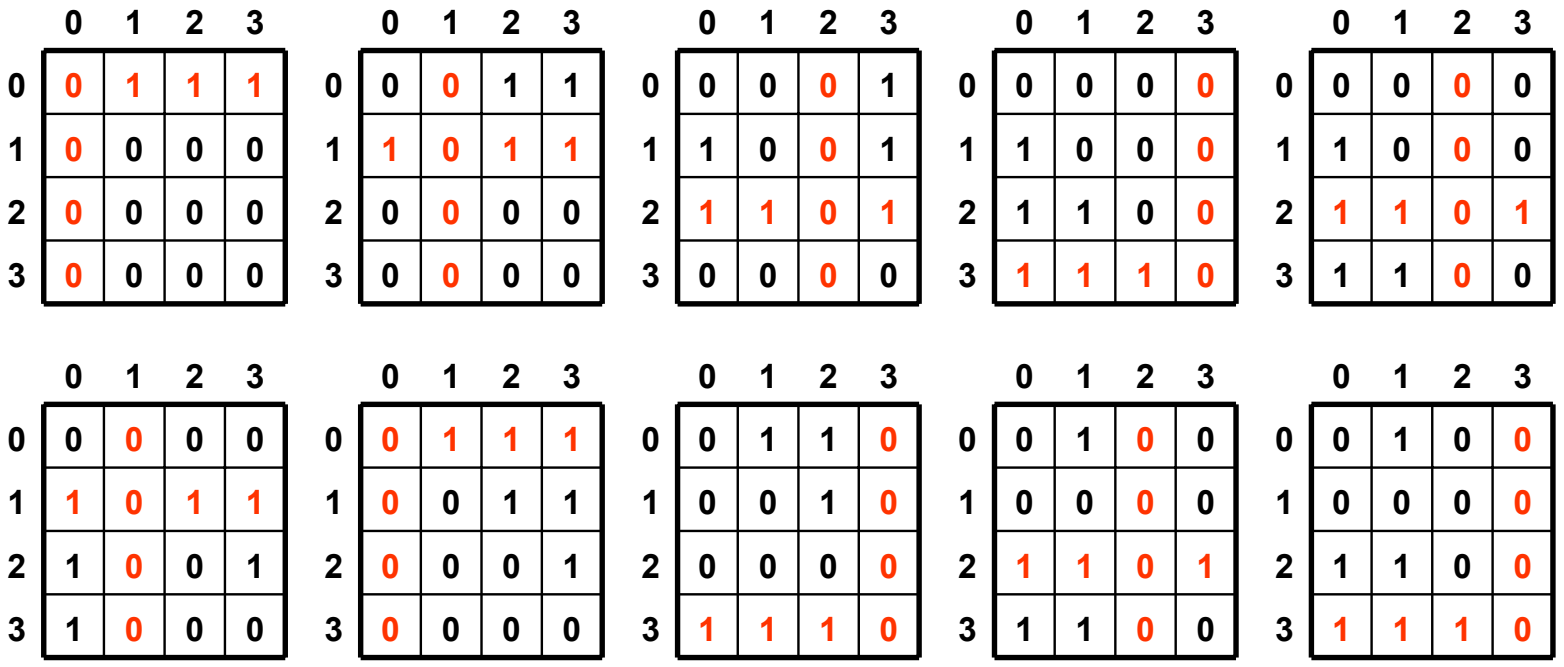
- V jakémkoliv okamžiku, **řádek s nejnižší binární hodnotou reprezentuje nejméně používanou stránku.**

Příklad: HW implementace LRU

Ke stránkám ve 4 rámcích bylo přistupováno v tomto pořadí:

0 1 2 3 2 1 0 3 2 3

Obrázky ukazují aktuální hodnoty v matici 4x4 po jednotlivých krocích:



LRU - SW implementace

- Algoritmy A) a B) jsou závislé na **speciálním hardwaru**.
- Můžeme použít softwarové řešení nazývané
  NFU (Not Frequently Used).

- S každou stránkou je spojen **SW čítač a bit *R***, na počátku nastavené na 0.

- Při přístupu na stránku je nastaven bit *R* na 1.

- Při přerušení od časovače
  - ***R* bit je přičten k čítači** u každé stránky a vynulován.
-
- Při výpadku stránky, **stránka s nejnižším čítačem je nahrazena.**

Aging Alg.

- **Problém:** Algoritmus NFU nic nezapomíná (nerozlišuje kdy se ke stránce přistupovalo) .

- **Řešení:** pomocí principu stárnutí.
  - Pří přerušení od časovače:
    - Obsah čítače je **posunut doprava** o jeden bit před přičtením bitu ***R***.
    - Bit ***R*** je přičten do **nejlevějšího bitu** čítače.
    - Bit ***R*** je vynulován.

- Při výpadku stránky, **nahradíme stránku s nejmenší hodnotou čítače.**

Příklad: Aging Alg.

clock tick 0	clock tick 1	clock tick 2	clock tick 3	clock tick 4
R bits for pages 0-5	R bits for pages 0-5	R bits for pages 0-5	R bits for pages 0-5	R bits for pages 0-5
1 0 1 0 1 1	1 1 0 0 1 0	1 1 0 1 0 1	1 0 0 0 1 0	0 1 1 0 0 0
Page				
0 10000000	11000000	11100000	11110000	01111000
1 00000000	10000000	11000000	01100000	10110000
2 10000000	01000000	00100000	00010000	10001000
3 00000000	00000000	10000000	01000000	00100000
4 10000000	11000000	01100000	10110000	01011000
5 10000000	01000000	10100000	01010000	00101000

Working Set Alg.

- Pro každý proces definujeme:
  - **Aktuální virtuální čas (*CVT*)** = množství času CPU, které proces skutečně využil.
  - **Pracovní množina (WS)** = množina stránek, ke kterým proces přistupoval během poledních τ jednotek svého *CVT*.

- **Každá položka** v tabulce stránek obsahuje
  - ***TLU*** – čas "posledního použití",
  - ***R*** – referenced bit,
  - ***M*** – modified bit.

- Při přístupu (čtení/zápis) ke stránce nastavíme bit ***R*** na 1.
- Při modifikaci stránky nastavíme ještě bit ***M*** na 1.
- **Periodicky při přerušení od časovače vynulujeme bit *R*.**

Working Set Alg. (2)

- **Při výpadku stránky**
  - Nechť ***AGE = CVT - TLU***.
  - Procházíme tabulku stránek a testujeme ***R*** bit:
    - Pokud (***R=1***), potom ***TLU=CVT*** and ***R=0***.
    - Pokud (***R=0 and AGE*** $> \tau$), potom nahradíme tuto stránku (nepatří do WS).
    - Pokud (***R=0 and AGE*** $\leq \tau$), potom si zapamatujeme stránku s největším ***AGE*** (tato stránka patří do WS).
  - Pokud všechny stránky patří do WS, **nahradíme stránku s největším *AGE*** (pokud jich bude více, vybereme náhodně jednu).
- **Nevýhoda:** musíme procházet celou tabulku stránek.

WSClock Alg.

- Lepší implementace working set algoritmu.
- **Informace o rámcích stránek** jsou uloženy v **kruhové frontě** orientované ve směru hodinových ručiček.
- Každá položka fronty obsahuje
  - ***TLU*** – čas “posledního použití”,
  - ***R*** – referenced bit,
  - ***M*** – modified bit.

WSClock Alg. (2)

- **Při výpadku stránky** je stránka, na kterou ukazuje ručička, testována jako první:
  - Pokud (***R***=1),
    potom ***TLU=CVT*** , ***R=0*** a posuneme ručičku.
  - Pokud (***R=0 and AGE*** $> \tau$ ***and M=0***),
    potom do tohoto rámce nahrajeme novou stránku.
  - Pokud (***R=0 and AGE*** $> \tau$ ***and M=1***)
    naplánujeme zápis obsahu rámce na disk a posuneme ručičku.
- **Pokud jsme otestovali celou frontu:**
  - Pokud **aspoň jeden zápis byl naplánován**, potom novou stránku nahrajeme do prvního uloženého rámce.
  - Pokud **žádný zápis nebyl naplánován**, potom všechny stránky patří do WS.
    - Pokusíme se použít první čistý rámec (***M*=0**).
    - Pokud neexistuje žádný čistý rámec, uložíme obsah aktuálního rámce a nahrajeme do něj novou stránku.

Shrnutí

Algoritmus	Vlastnosti
Optimal	Slouží pouze jako benchmark
NRU (Not Recently Used)	Velmi hrubý
FIFO (First-In, First-Out)	Nahradí i důležité stránky
Second chance	Výrazné vylepšení FIFO
Clock	Realistický
LRU (Least Recently Used)	Výborný, těžko se implementuje
NFU (Not Frequently Used)	Hrubá aproximace LRU
Aging	Efektivní aproximace LRU
Working set	Výborný, špatná implementace
WSClock	Efektivní algoritmus

Windows XP

- **32 bitový x86**
  - **VAS**
    - 2GB User Space + 2GB System Space
    - 3GB User Space +1GB System Space
  - **Velikost stránek**
    - Malé 4KB
    - Velké 4MB (2Mb na systémech s Physical Address Extension PEA)
  - **Překlad VA na FA**
    - Dvouúrovňová tabulka stránek + TLB
    - Tříúrovňová tabulka stránek + TLB (na PEA)

Windows XP (2)

- **64 bitový x64**
  - **VAS**
    - 8192GB (8TB) User Space + 6657GB System Space
  - **Velikost stránek**
    - Malé 4KB
    - Velké 2MB
  - **Překlad VA na FA**
    - Čtyřúrovňová tabulka stránek + TLB

- **Working set**
  - Množina stránek, ke kterým přistupují vlákna daného procesu

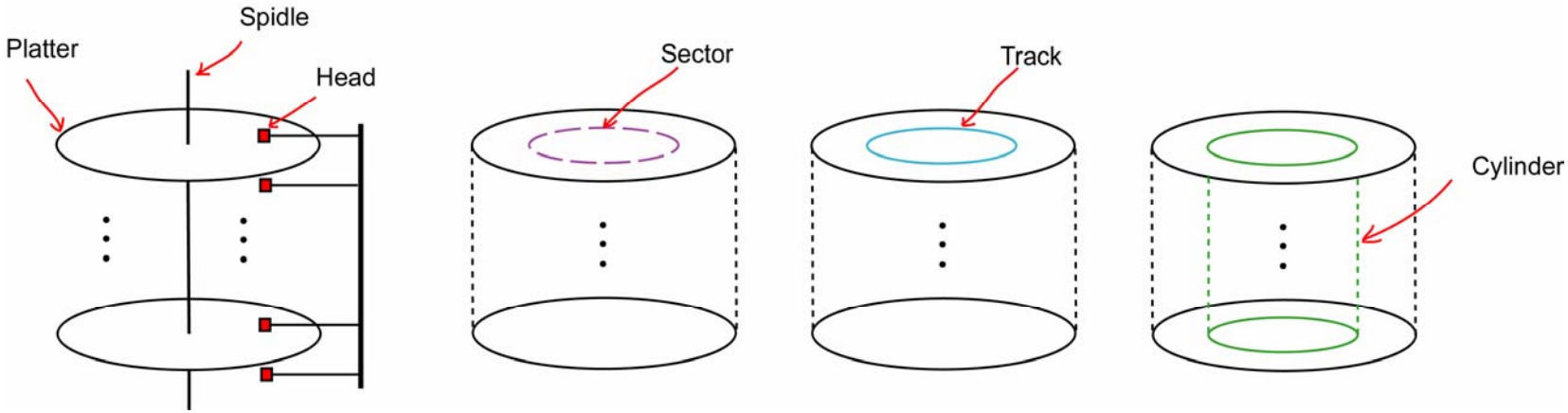
- **Nahrávání stránek**
  - Při spuštění: prepaging
  - Po nahrání: demand paging

- **Náhrada stránek**
  - Aging alg.
  - Systém udržuje dostatečný počet volných rámců.
  - Periodicky provádí náhradu a ukládání stránek.

Operační systémy

Přednáška 10: Uložení dat – disky, RAID

Fyzická geometrie disku



- **Sektor (Sector)**
  - Nejmenší adresovatelná jednotka na disku (obvykle 512B).
- **Stopa (Track)**
  - Množina sektorů na jednom povrchu ve stejné vzdálenosti od středu.
  - Počet sektorů ve stopě se může lišit v závislosti na poloměru.
- **Cylindr (Cylinder)**
  - Množina všech stop o daném poloměru.

Virtuální geometrie disku

- **Staré disky**
  - Počet sektorů na stopu byl stejný ve všech cylindrech.
- **Moderní disky**
  - Jsou rozděleny na zóny (uvnitř zóny je počet sektorů na stopu konstantní, ve vnitřních zónách je menší počet sektorů na stopu než ve vnějších).
  - **Virtuální geometrie (CHS adresování)**
    - SW předpokládá, že počet sektorů ve stopě je konstantní.
    - Adresa sektoru na disku se skládá ze tří čísel: ***C, H, S***.
  - **Fyzická geometrie**
    - Řadič disku mapuje požadavek (***C,H,S***) na reálné číslo cylindru, hlavičky a sektoru.
  - **Logical block addressing (LBA)**
    - Diskové sektory jsou číslovány sekvenčně od 0, bez ohledu fyzickou geometrii.

Přístup na disk

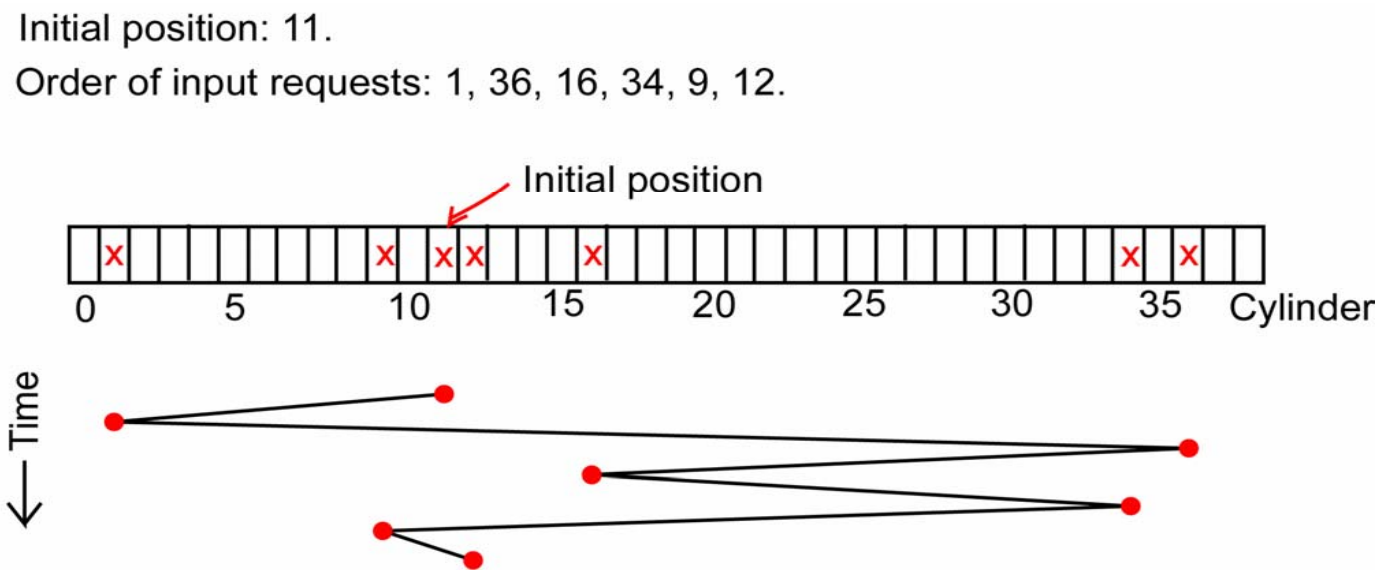
- **Jak dlouho trvá zápis/čtení jednoho sektoru?**
  - **doba vystavení (seek time)**
    - čas nastavení hlaviček na správný cylindr
  - **rotační zpoždění (rotational delay)**
    - čas posunutí správného sektoru pod hlavičku
  - **čas přenosu dat**

- **OS je odpovědný za efektivní používání disků**
  - **rychlý přístup** (minimalizování doby vystavení)
  - **velká šířka pásma** (maximalizace počtu přenesených bytů za čas)

- **Algoritmy plánování přístupu na disk**
  - dříve implementované v OS, nyní v řadičích disků
  - určují pořadí zpracování jednotlivých požadavků (např. NCQ pro SATA disky)

First-In-First-Out (FIFO)

- Ovladač disku přijímá V/V požadavky a obsluhuje je v pořadí v jakém přišly.
- **Výhody**: spravedlnost
- **Nevýhody**: horší výkon

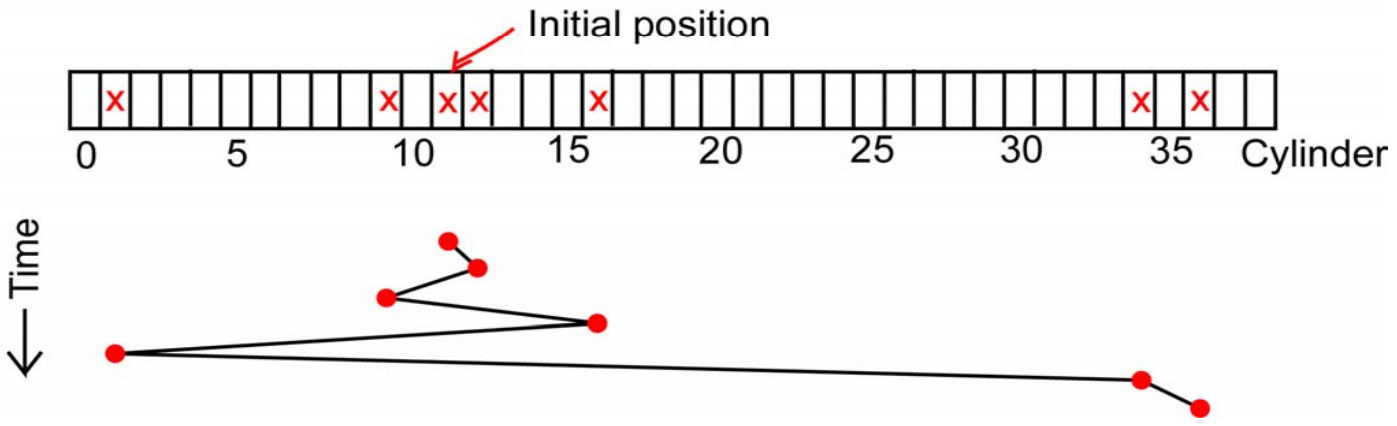


Shortest Service Time First (SSTF)

- Nejdříve jsou obslouženy požadavky z fronty, které vyžadují nejmenší pohyb hlaviček z aktuální pozice.
- **Výhody**: lepší výkon než FIFO
- **Nevýhody**:
  - hlavičky mají tendenci setrvávat uprostřed disku
  - vzniká problém stárnutí u V/V požadavků z krajních pozic

Initial position: 11.

Order of input requests: 1, 36, 16, 34, 9, 12.

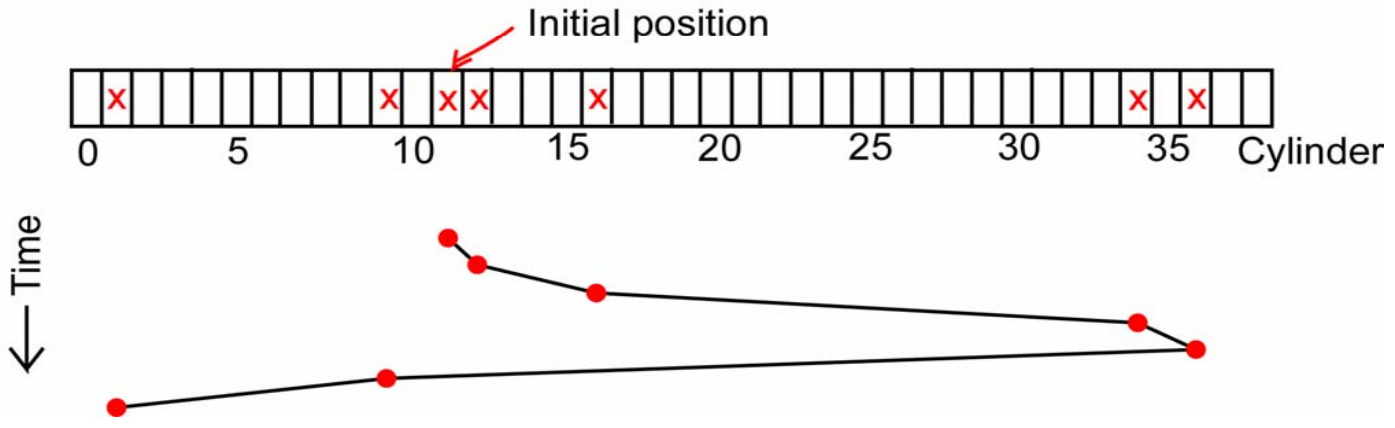


SCAN algorithm (elevator algorithm)

- Hlavičky se pohybují nejdříve jedním směrem a uspokojí se všechny požadavky v daném směru.
- Pokud už není žádný požadavek v daném směru, směr se změní a uspokojují se zbylé požadavky.
- **Výhoda**: odstranil se problém stárnutí.
- **Nevýhoda**: trochu horší výkon než SSTF

Initial position: 11.

Order of input requests: 1, 36, 16, 34, 9, 12.



N-step SCAN

- Fronta požadavků je rozdělena na podfronty délky *N.*
- Jednotlivé podfronty jsou zpracovány najednou algoritmem SCAN.

- **Výhoda:** snižuje možnost ustrnutí hlaviček nad hodně vytěžovanými cylindry.

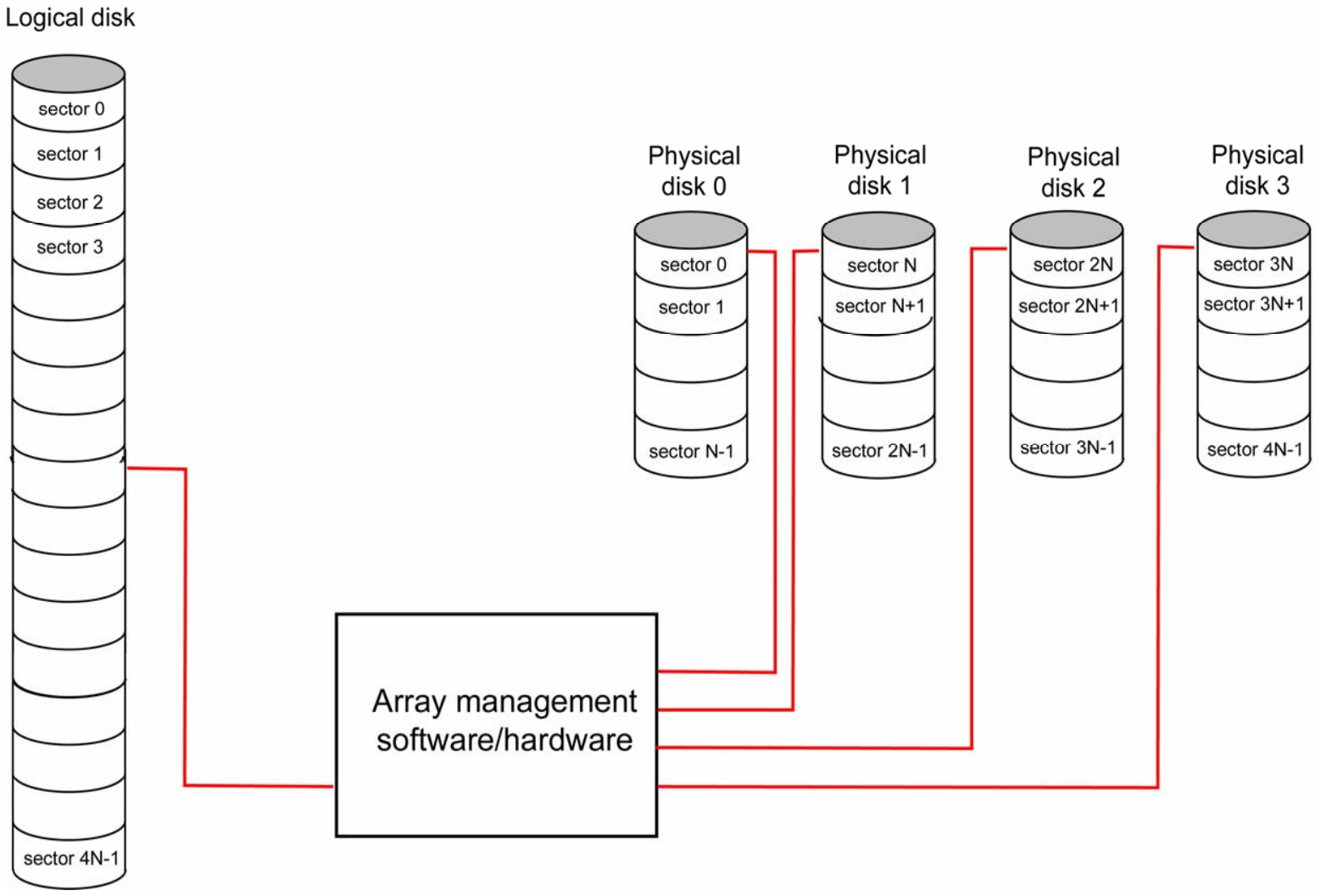
Redundant Array of Independent Disks (RAID)

- 1988, University of California at Berkeley
- **SLED** = Single Large Expensive Disk
  - Mean Time To Failure (MTTF) = střední doba do poruchy disku
- **RAID** = Redundant Array of Independent (Inexpensive) Disks
  - RAID je množina fyzických disků, které OS vidí jako **jeden logický disk**.
  - **data jsou distribuována** mezi jednotlivé fyzické disky
  - **část (redundantní) diskové kapacity** slouží pro uložení pomocných informací (např. parita,…) nutných pro obnovu v případě poškození některých disků
  - Mean Time To Failure (MTTF) – střední doba poškození disku
  - MTTF pole RAID = MTTF jednoho disku/počet disků v RAIDu.

RAID (2)

- **Hardwarový RAID**
  - RAID SCSI řadič + pole SCSI disků
  - RAID se jeví pro OS jako jeden velký disk
- **Softwarový RAID**
  - RAID software + pole SCSI disků
  - RAID software může být součástí OS
  - RAID se jeví pro OS jako jeden velký disk
- **Výhody**
  - vysoká spolehlivost,
  - velká šířka pásma (počet přenesených bytů/doba od zadaní do ukončení požadavku) .

RAID 0 - zřetězení



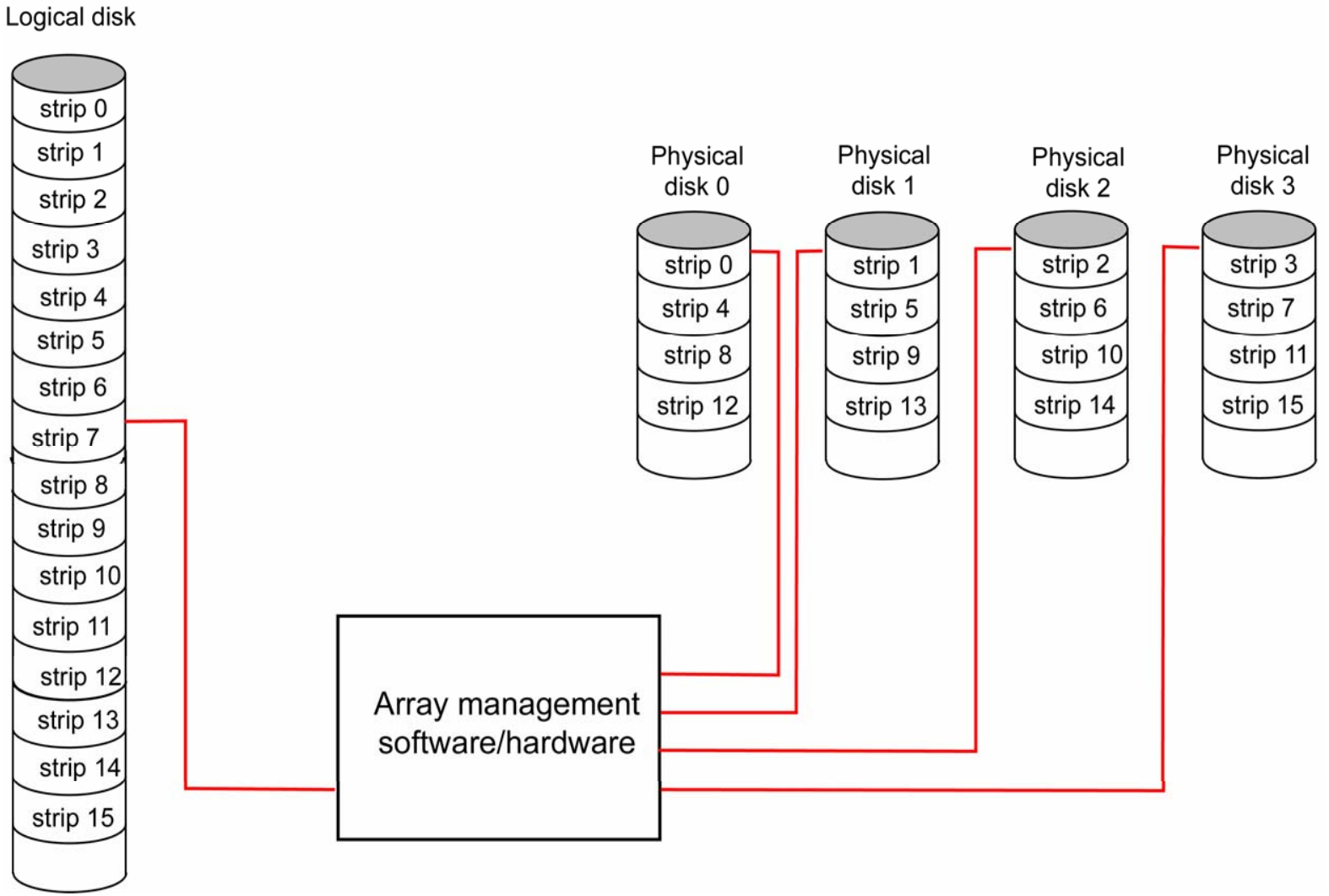
RAID 0 – zřetězení (2)

- Data jsou **postupně zapisována** na jednotlivé disky (po naplnění prvního disku, pokračujeme na druhém,…)
- Operace čtení/zápis stejně rychlé jako u jednoho disku.
- V SW RAIDu slouží jako přípravný krok pro zrcadlení (např. již existujícího systémového FS)
- **Výhody**
  - Získáme velký logický disk.
  - Lze využít 100% diskové kapacity.
- **Nevýhody**
  - **Žádná redundance** ⇒ poškození jednoho disku v poli způsobí ztrátu všech dat.

RAID 0 - stripování

- Data jsou dělena (stripována) po blocích (stripech) mezi jednotlivé disky.
- Logický disk se skládá ze stripů.
- **strip** = množina *k* po sobě následujících sektorů.

RAID 0 - stripování (2)

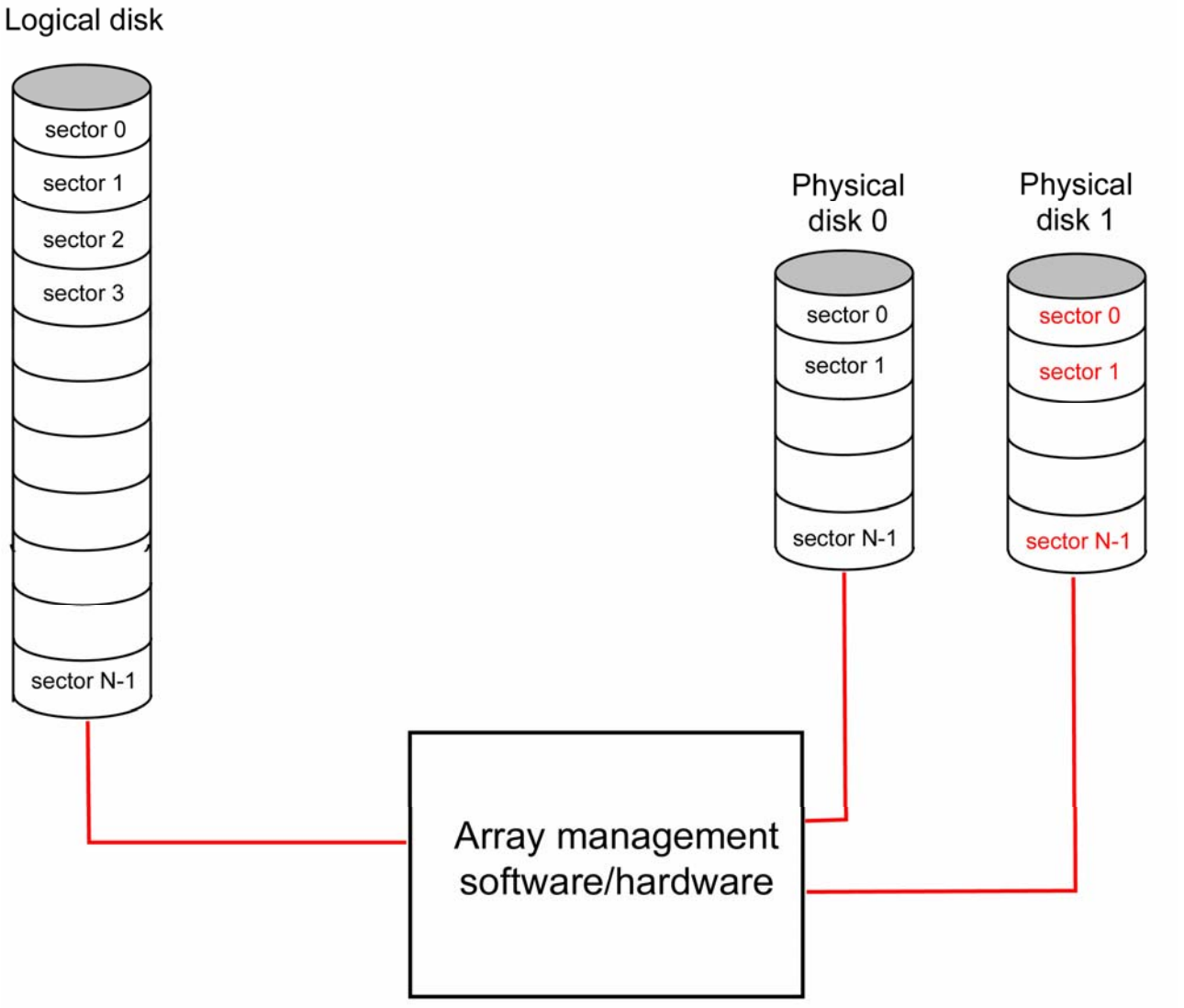


RAID 0 - stripování (3)

- **Nevýhody**
  - **Žádná redundance** ⇒ poškození jednoho disku v poli způsobí ztrátu všech dat.

- **Výhody**
  - lze využít 100% diskové kapacity,
  - **zvýšení propustnosti** vyvážením zátěže malými přístupy,
  - paralelizace velkých přístupů s cílem **zkrácení doby odpovědi**.

RAID 1 – zrcadlení

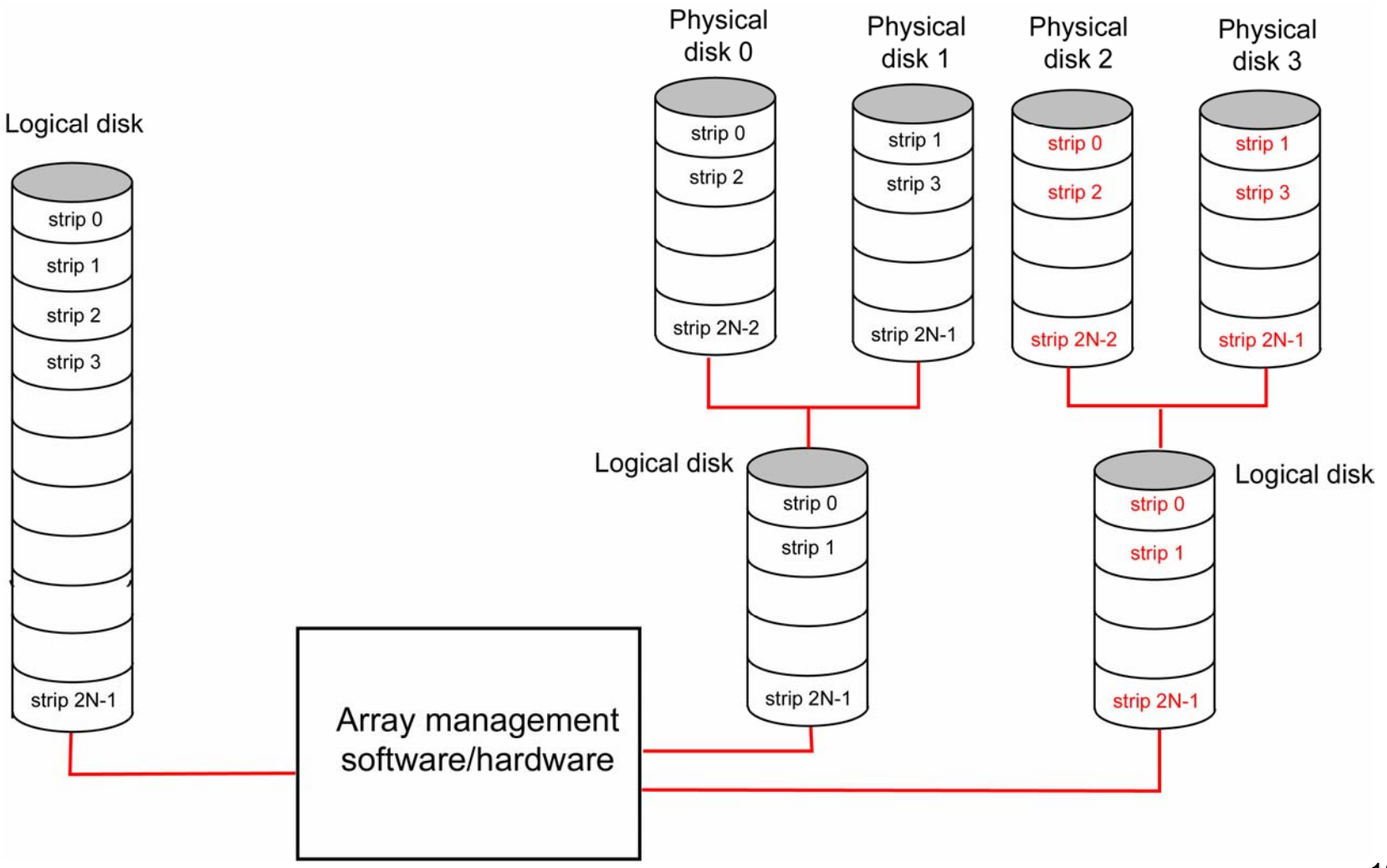


RAID 1 – zrcadlení (2)

- **Nevýhody**
  - Využijeme pouze 50% diskové kapacity.

- **Výhody**
  - Při ztrátě disku, lze data okamžitě získat z kopie.
  - **Zápis** může být **stejně rychlý** jako u jednoho disku (podle strategie zápisu: paralelní/sériová).
  - **Čtení** dat může být je **rychlejší** než u jednoho disku (podle strategie čtení: round robin/geometric/first).

RAID 0+1 stripování

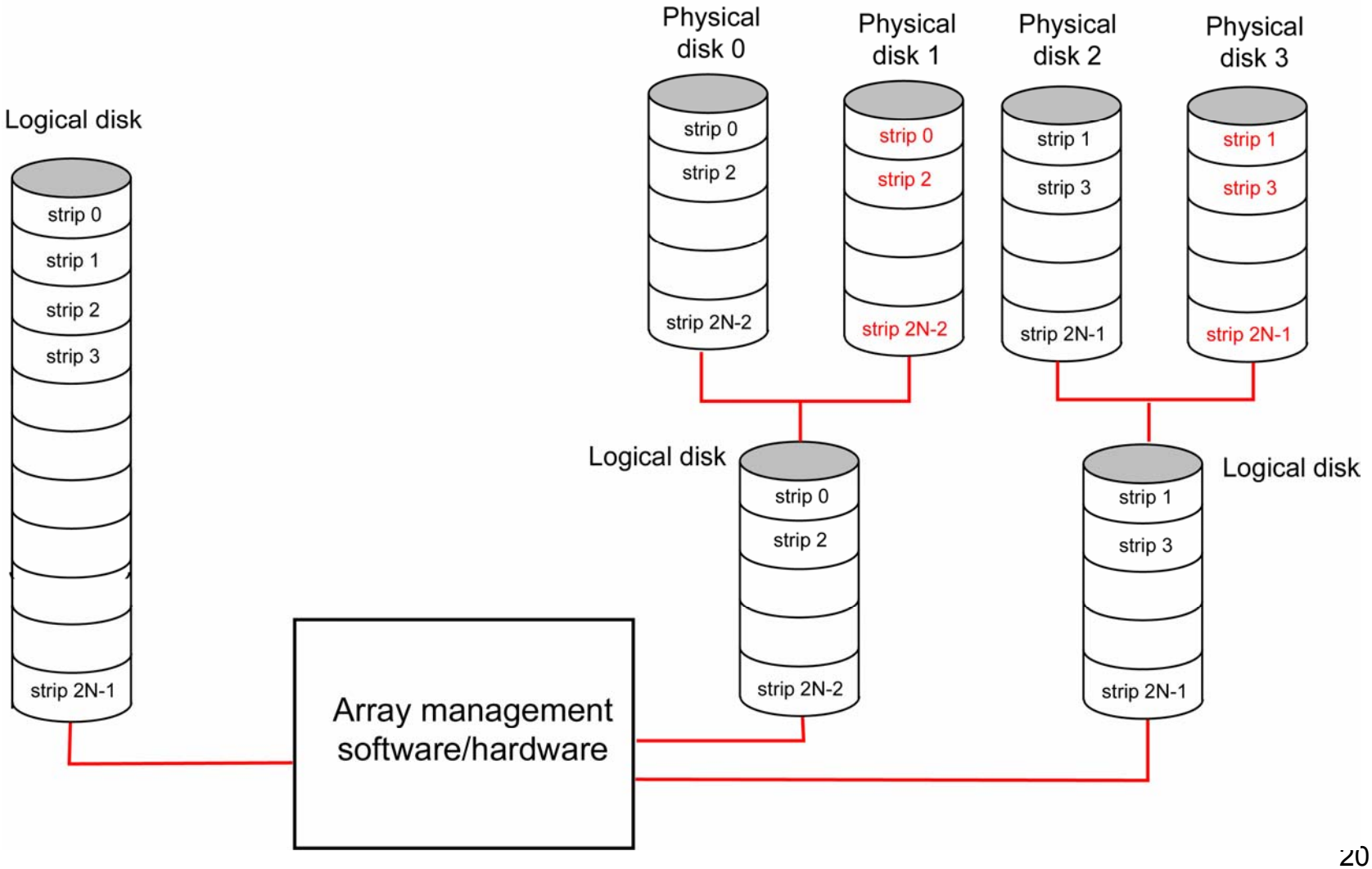


RAID 0+1 stripování

- **Výhody**
  - data distribuovaná mezi disky,
  - čtení/zápis může být rychlejší než u jedno disku,
  - redundance.

- **Nevýhody**
  - lze využít pouze 50% diskové kapacity,
  - výpadek jednoho disku způsobí ztrátu redundance.

RAID 1+0 stripování



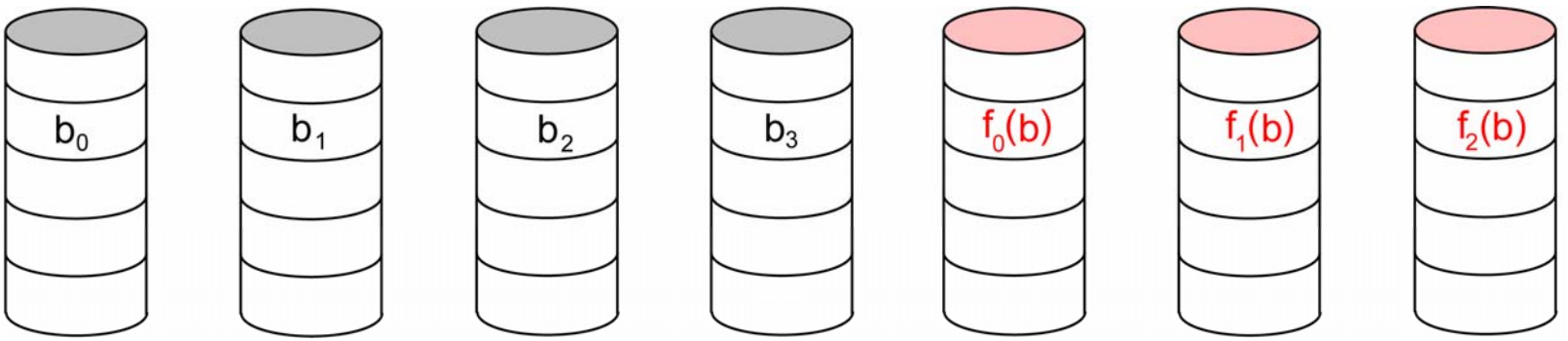
RAID 1+0 stripování

- **Výhody**
  - podobné jako u RAID 0+1 stripování,
  - plus navíc
    - toleruje větší počet poškozených disků,
    - rychlejší obnova dat.

- **Nevýhody**
  - lze využít pouze 50% diskové kapacity.

RAID 2

- Všechny disky jsou **synchronizovány**.
- **Data jsou stripována po bytech** mezi jednotlivé disky.
- **Zabezpečení dat pomocí Hammingova kódu** (correct single-bit errors and detect double-bit errors).
- Počet redundantních disků je úměrný počtu datových disků.



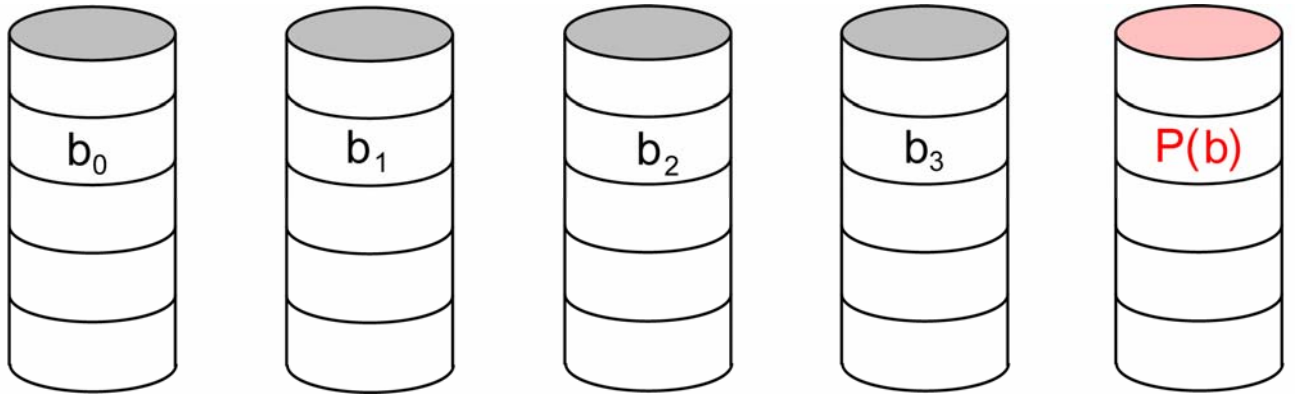
RAID 2 (2)

- **Nevýhody**
  - lze využít více než 50% diskové kapacity,
  - malá propustnost.

- **Výhody**
  - paralelizace velkých přístupů s cílem zkrácení doby odpovědi.

RAID 3

- Zjednodušená verze RAID 2.
- **Pro zabezpečení se požívá parita** uložená na paritním disku ($P(b)=b_0$ XOR b_1 XOR b_2 XOR b_3).



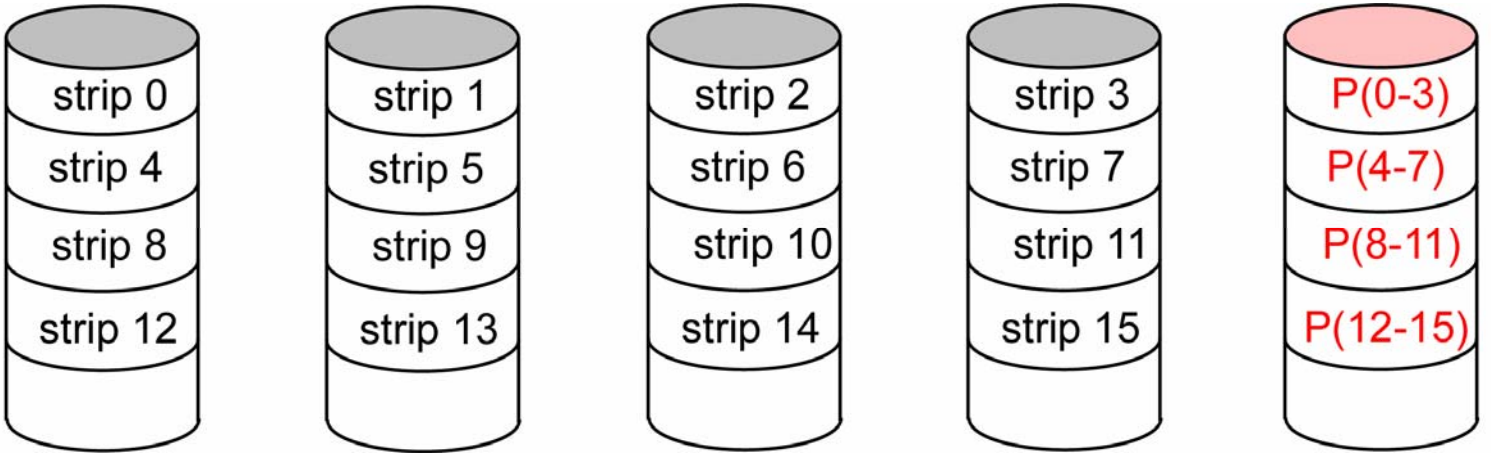
RAID 3 (2)

- **Nevýhody**
  - malá propustnost.

- **Výhody**
  - pouze jeden disk obsahuje paritní informace, ostatní lze využít pro data.
  - paralelizace velkých přístupů s cílem zkrácení doby odpovědi.

RAID 4

- Disky nejsou synchronizovány jako u RAID 2 a 3.
- Používá se **stripování po blocích (stripech).**
- Pro zabezpečení se požívá **parita po stripech**, která je uložená na paritním disku.



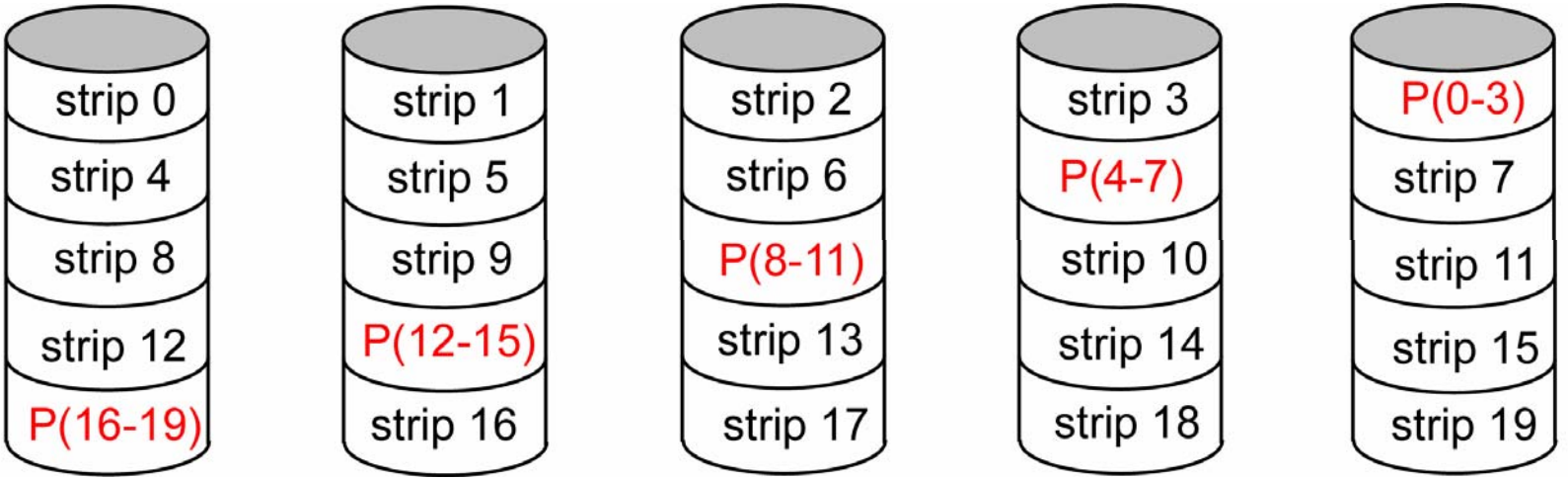
RAID 4 (2)

- **Nevýhody**
  - zápis dat,
  - přetížení paritního disku.

- **Výhody**
  - pouze jeden disk obsahuje paritní informace, ostatní lze využít pro data.
  - paralelizace čtení velkých přístupů s cílem zkrácení doby odpovědi.

RAID 5

- RAID 5 je organizován podobně jako RAID 4, ale **parita je distribuována** mezi všemi disky.



RAID 5 (2)

- **Nevýhody**
  - zápis dat.

- **Výhody**
  - pouze jeden disk obsahuje paritní informace, ostatní lze využít pro data.
  - paralelizace čtení velkých přístupů s cílem zkrácení doby odpovědi.

- Vhodný pokud počet operací zápisu nepřekročí 15 až 20%.

Připojení disků

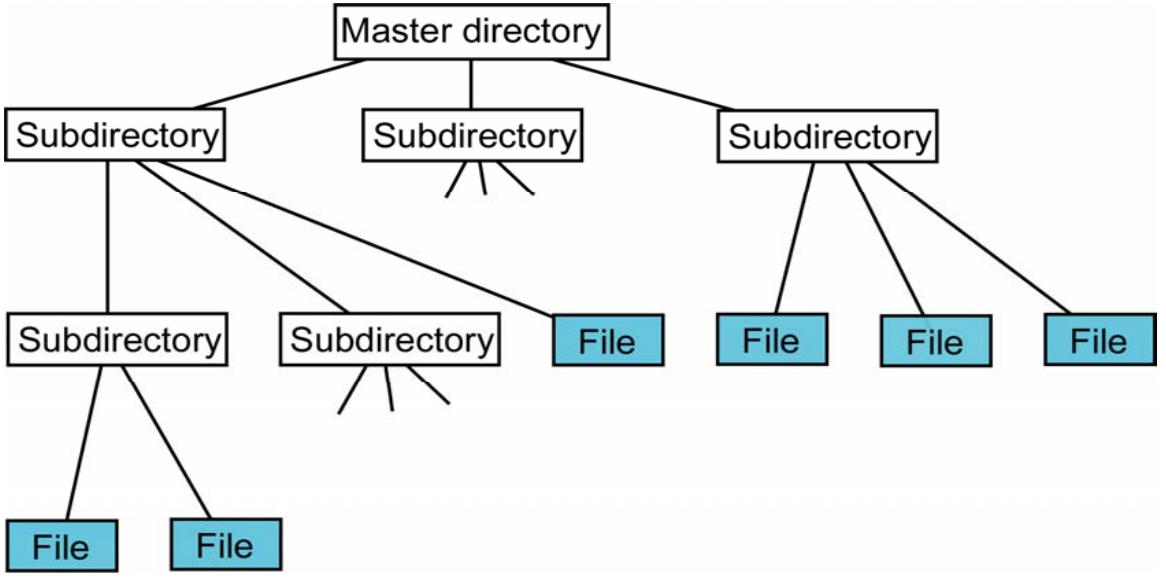
- **Přímo k počítači** přes V/V porty
  - **Direct Attached Storage (DAS)**
    - SCSI (Small Computer System Interconnect)
    - ATA/SATA

- **Prostřednictvím sítě**
  - **Network-Attached Storage (NAS)**
    - Ethernet,…
  - **Storage-Area Network (SAN)**
    - FC (Fibre channel), iSCSI (Internet SCSI), FCIP (Fibre Channel over Internet Protocol), …

Operační systémy

Přednáška 11: Souborové systémy

Strom adresářů



- **Soubory**
  - umožňují uložení informace na disku..
- **Adresáře/Složky**
  - umožňují hierarchické uspořádání dat na disku.

Soubory

Soubor = jméno + atributy + data

- **Jméno souboru**
  - Délka (8+3, 255,…), kódování (ASCII, UNICODE,…).
- **Atributy souboru**
  - Typ souboru (adresář, obyčejný soubor, link,…).
  - Vlastníci souboru (uživatel, skupina, ostatní,…).
  - Přístupová práva (čtení, zápis, spuštění, setuid, ACL,…).
  - Čas (vytvoření, modifikace, přístupu,…). …
- **Data**
  - Obsah souboru, který je uložen v datových blocích na disku.

- **Přístup k souboru**
  - Pomocí systémových volání: open(), close(), seek(), read(), write(), stat(),...
  - Příkazy OS, aplikace.

Adresáře/Složky

- Umožňují **hierarchické uložení informací** ve stromě adresářů.

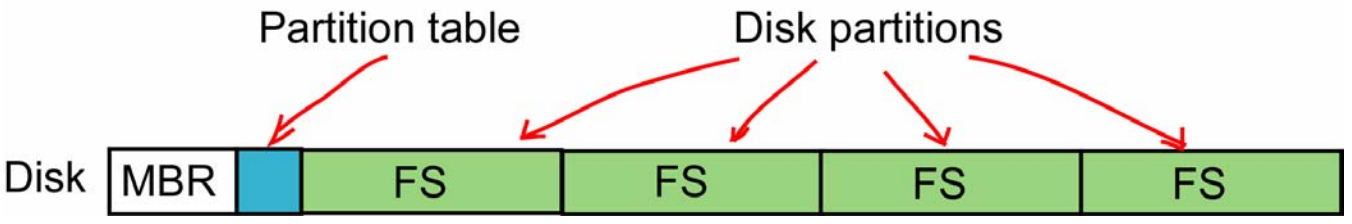
- **Cesta k souboru/adresáři**
  - **Absolutní**
    - Začíná vždy v kořenovém adresáři *root*.
    - Obsahuje posloupnost podadresářů mezi *root* a cílovým souborem.

      /home/rocnik2/skupuna12/Novak

  - **Relativní**
    - Začíná vždy v aktuálním adresáři *current*.
    - Obsahuje posloupnost podadresářů mezi *current* a cílovým souborem.

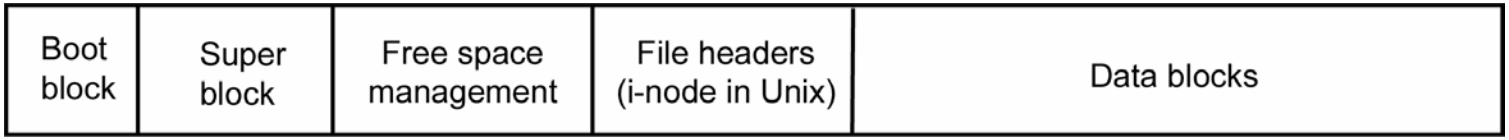
      current=/home/rocnik2/skupina15
      ../skupina12/Novak
- **Přístup k adresáři**
  - Pomocí systémových volání: opendir(), creat(), getfirts(), getnext(), link()…
  - Příkazy OS, aplikace.

Rozvržení disku



- **Master Boot Record (MBR)**
  - Obsahuje program pro zavedení jádra OS z disku do paměti.
- **Tabulka oblastí (Partition table)**
  - Disk může být rozdělen **na několik diskových oblastí.**
  - Každá oblast může obsahovat svůj systém souborů.

Rozvržení systému souborů (FS)



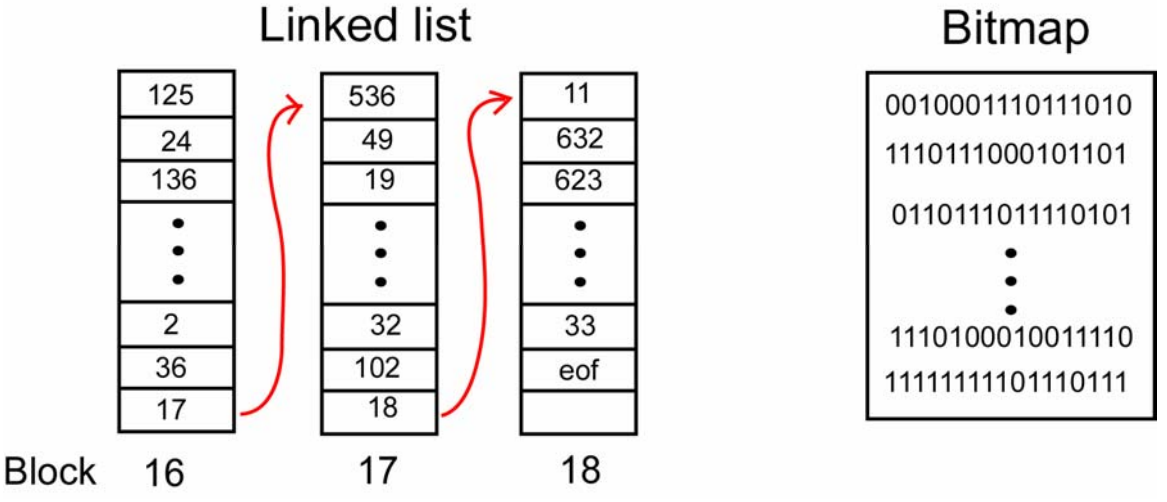
- **Super block** obsahuje klíčové informace o FS:
  - Magické číslo (pro identifikaci typu FS),
  - Velikost datových bloků a jejich počet v daném FS,
  - Velikost "File headers" oblasti,
  - Umístění kořenového adresáře, …
- **Free space management**
  - Informace o volných datových blocích a „file headers".
- **File headers**
  - Pole struktur (jedna na soubor), obsahujících např. atributy souborů a adresy datových bloků.
- **Datové bloky**
  - Je v nich uložen obsah adresářů a souborů.

Sector vs. datový blok

- **Sector** (fyzická jednotka)
  - Minimální adresovatelná jednotka na disku (typicky 512B).
- **Datový blok** (logická jednotka)
  - Minimální adresovatelná jednotka ve FS (souvislá posloupnost sektorů).
- **Příklad**:
  - UNIX **ufs** má 4kB, 8kB, 16kB, 32kB, 64kB datové bloky.
  - UNIX **vxfs** má 1 kB datové bloky.
  - MS Windows **FAT32** má 4KB,...,32KB datové bloky.
    ntfs má 512B,...,64KB datové bloky.
- **Velké dat. bloky**
  - Vhodné pro FS s velkými soubory a soubory s sekvenčním přístupem.
- **Malé dat. bloky**
  - Vhodné pro FS s malými soubory a soubory s náhodným přístupem.

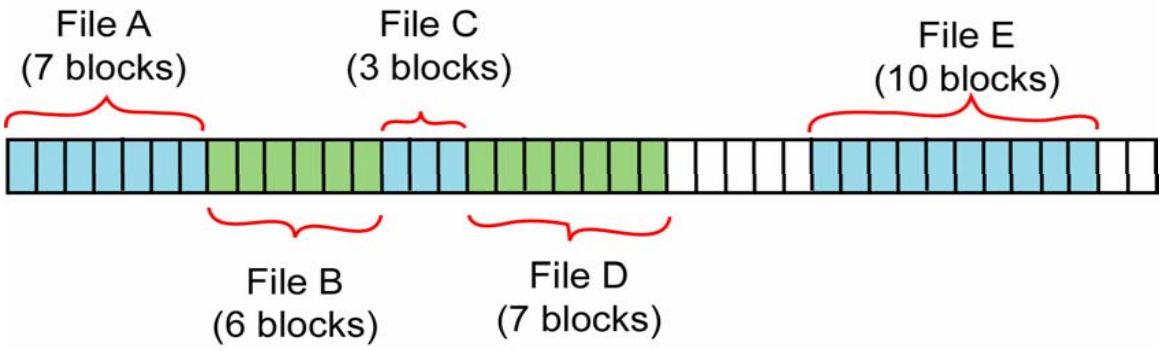
Správa volného prostoru

- **Zřetězený seznam**
  - Je uložen ve volných datových blocích.
  - V hlavní paměti je pouze část tohoto seznamu.
- **Bitová mapa**
  - Většinou zabírá méně místa než zřetězený seznam.
  - Pouze v případě téměř zaplněného FS je zřetězený seznam výhodnější.



Souvislá alokace dat. bloků

- Obsah souboru je uložen na disku v **souvislé posloupnosti dat. bloků.**
- Používá se např. v UNIX Veritas File System (vxfs).



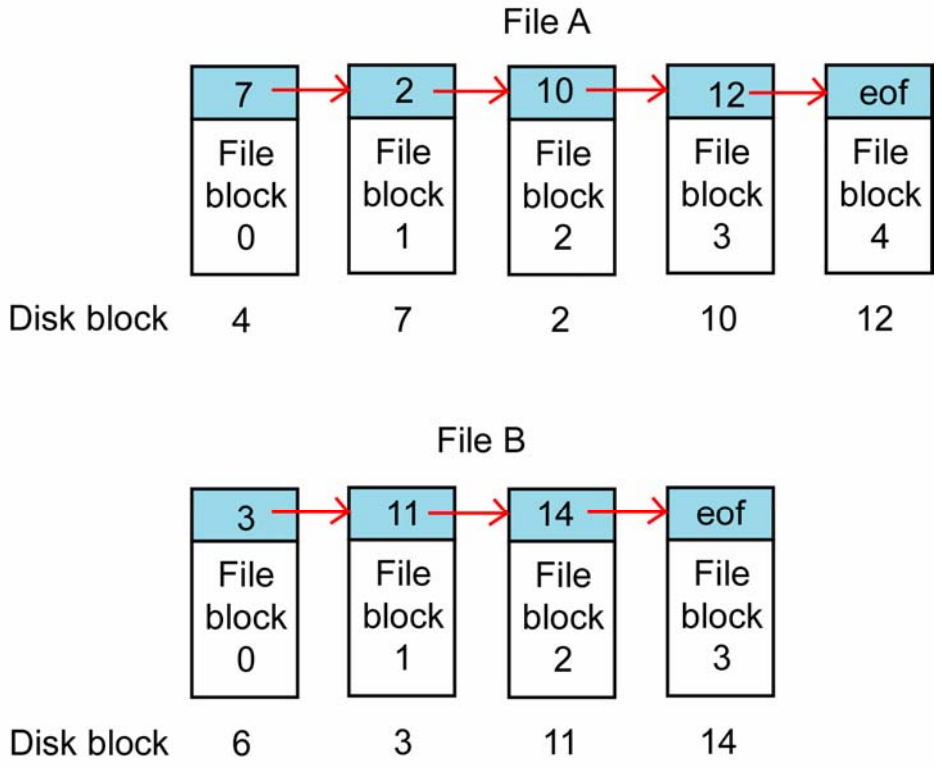
Souvislá alokace dat. bloků (2)

- **Výhody**
  - **Jednodušší implementace**.
  - Pro přístup k obsahu souboru musíme znát pouze:
    - diskovou adresu prvního dat. bloku,
    - počet dat. bloků alokovaných pro soubor.
  - **Výborný výkon čtení/zápisu**.

- **Nevýhody**
  - **Složitější alokace dat. bloků** (obvykle naznáme maximální velikost souboru při jeho vytváření).
  - **Fragmentace** disku.

Alokace dat. bloků pomocí zřetězeného seznamu

- Každý dat. blok obsahuje data a ukazatel na následující dat. blok.

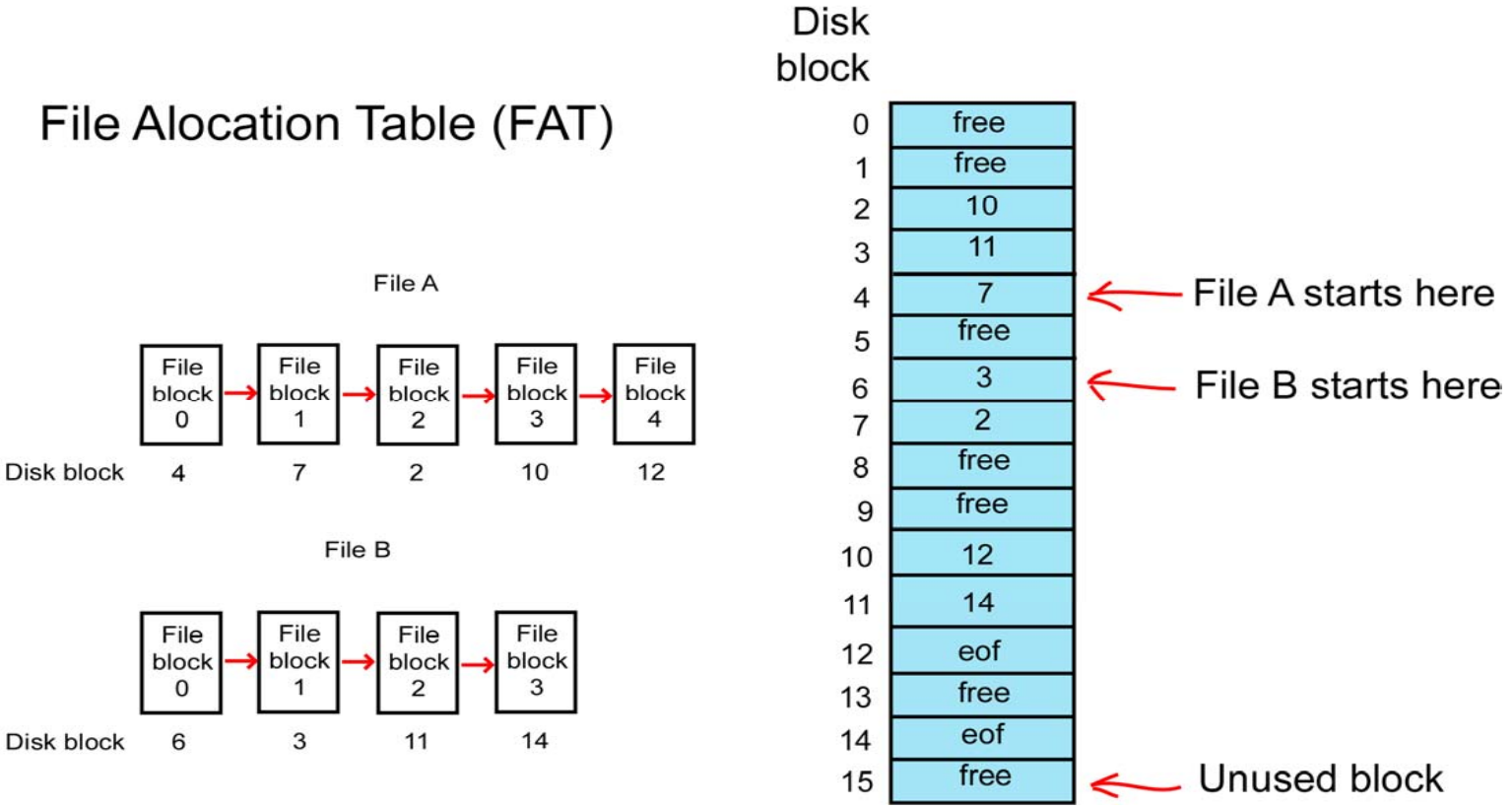


Alokace dat. bloků pomocí zřetězeného seznamu (2)

- **Výhody**
  - Pro přístup k obsahu souboru musíme znát pouze diskovou adresu prvního dat. bloku.
  - **Žádná fragmentace**.
- **Nevýhody**
  - **Pomalý náhodný přístup**.
  - Množství **dat v dat. bloku není přesně mocninou dvou.**

Alokace dat. bloků pomocí tabulky

- Alokace dat. bloků je založená na **zřetězeném seznamu**, ale **ukazatelé** na následující dat. blok **jsou uloženy v tabulce FAT**.
- Při přístupu k FS je FAT nebo její část nahrána do hlavní paměti.



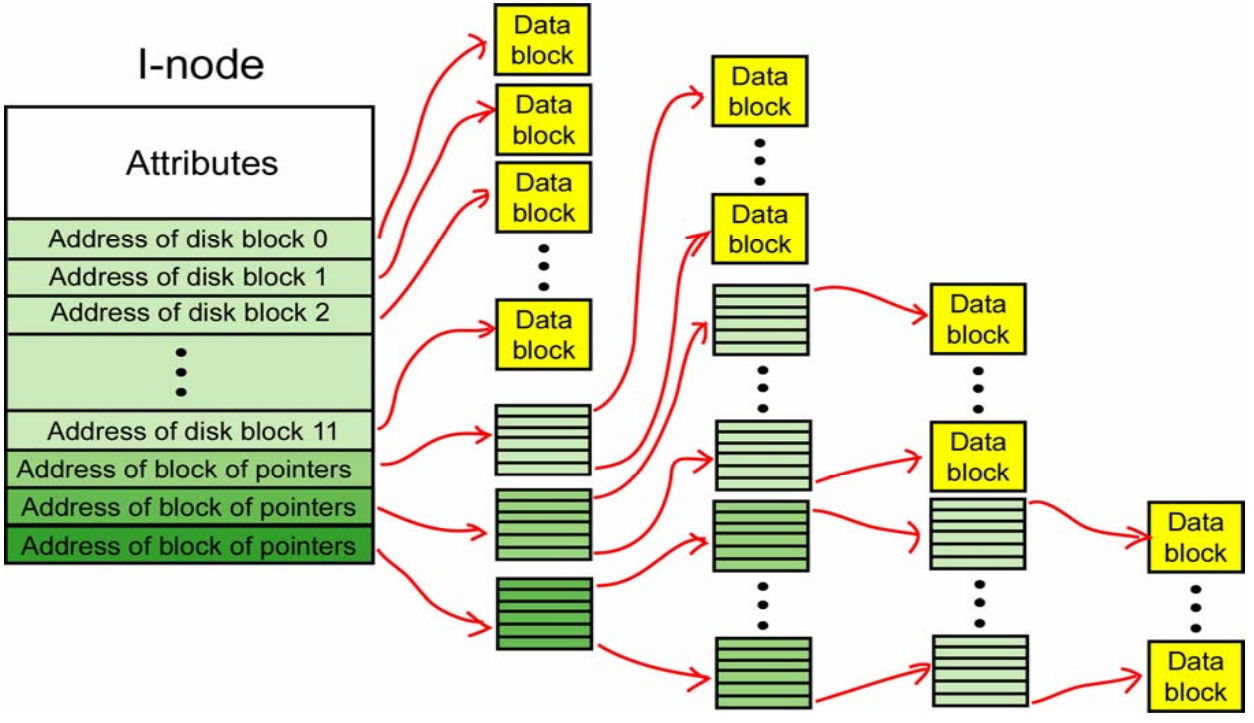
Alokace dat. bloků pomocí tabulky (2)

- **Výhody**
  - Celý dat. blok je využit pro uložení dat.
  - Náhodný přístup je rychlý (pokud je FAT v hlavní paměti).
  - Pro přístup k souboru stačí znát adresu prvního dat. bloku.
  - Informace o volných dat. blocích je obsažená ve FAT.

- **Nevýhody**
  - Velikost FAT (např. pro 20GB FS s velikostí dat. bloku 1KB, má FAT 20 milionů položek) .
  - Celá/část FAT musí být v hlavní paměti během používání FS.

Index nodes (I-nodes)

- i-node je struktura, která obsahuje jak **atributy souboru**, tak **adresy datových bloků**, ve kterých je uložen obsah souboru.
- Jsou tam **přímé adresy** a **nepřímé adresy** (první, druhé a třetí úrovně).
- **Speciální dat. bloky** jsou určeny pro uložení adres datových bloků při nepřímé adresaci.



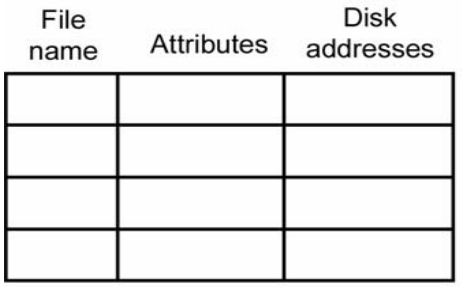
I-nodes (2)

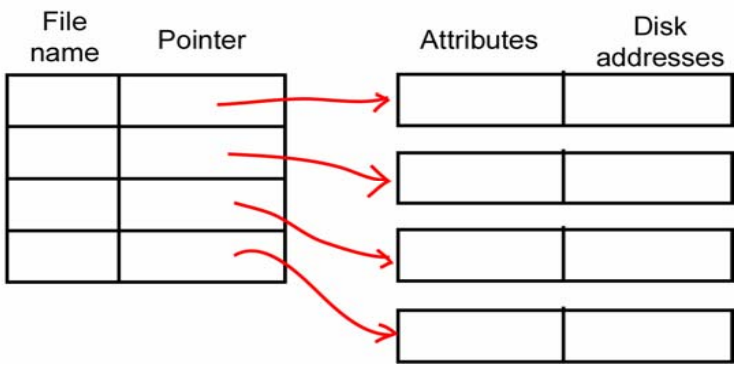
- **Výhody**
  - V hlavní paměti jsou pouze i-nodes otevřených souborů.
- **Nevýhody**
  - Přístup k velkým souborům je pomalejší než k malým souborům.
  - Přibližně 10% datových bloků ve FS je použito na uložení adres dat. bloků (speciální dat. bloky).

Implementace adresářů

- **Struktura položky adresáře**
  1. **jméno souboru, atributy souboru a adresy dat. bloků souboru**
     (FAT12, FAT16, FAT32).

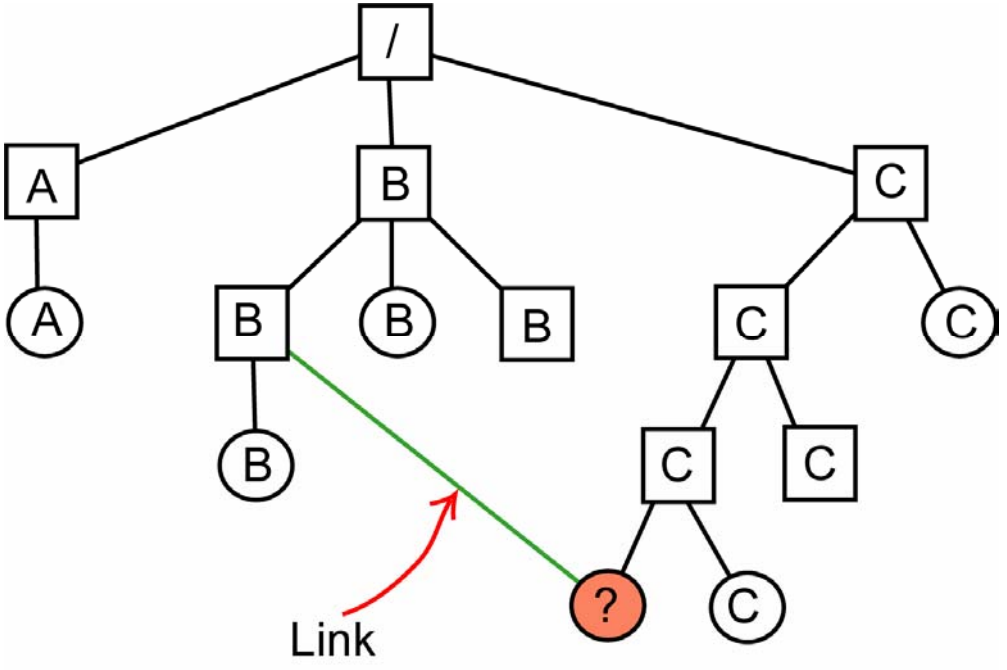
  2. **jméno souboru a ukazatel na speciální strukturu**, která obsahuje atributy souboru a adresy dat. bloků souboru (např. UNIX).





Sdílené soubory/adresáře

- Sdílené soubory mohou **být viděny současně** z různých adresářů pod různými jmény.
- **Změny obsahu souboru** v jednom adresáři by měli být **viděny ve všech adresářích**.
- **Link** = spojení mezi sdíleným souborem a adresáři.



Sdílené soubory (2)

- **Problém**
  - Nechť v adresářích jsou uloženy i adresy dat. bloků souboru.
  - Pokud jeden uživatel zvětší sdílený soubor o jeden dat. blok, pak ostatní uživatelé tuto změnu neuvidí.
- **Řešení 1** (hard link)
  - Každému souboru přiřadíme strukturu, která bude obsahovat adresy dat. bloků daného souboru (např. i-node).
  - V adresáři bude ukazatel na tuto strukturu..
- **Řešení 2** (soft link)
  - Adresáře ukazují na systémový soubor typu „LINK“, který obsahuje cestu ke sdílenému souboru.

Příklad: MS-DOS

- Používá alokaci datových bloků **pomocí tabulky FAT (File Allocation Table)**

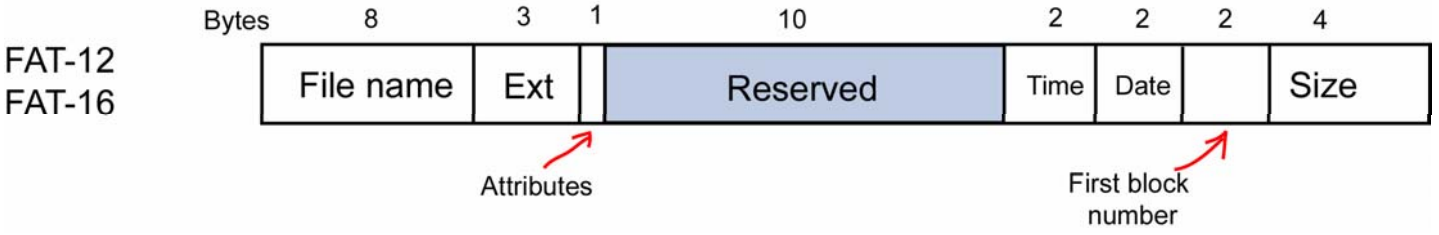
- **Jména souborů**
  - 8+3 velkých znaků: jméno+přípona (FAT-12, FAT-16),
  - 256 znaků (FAT-32).

- Alokace soborů probíhá po datových blocích (cluster).

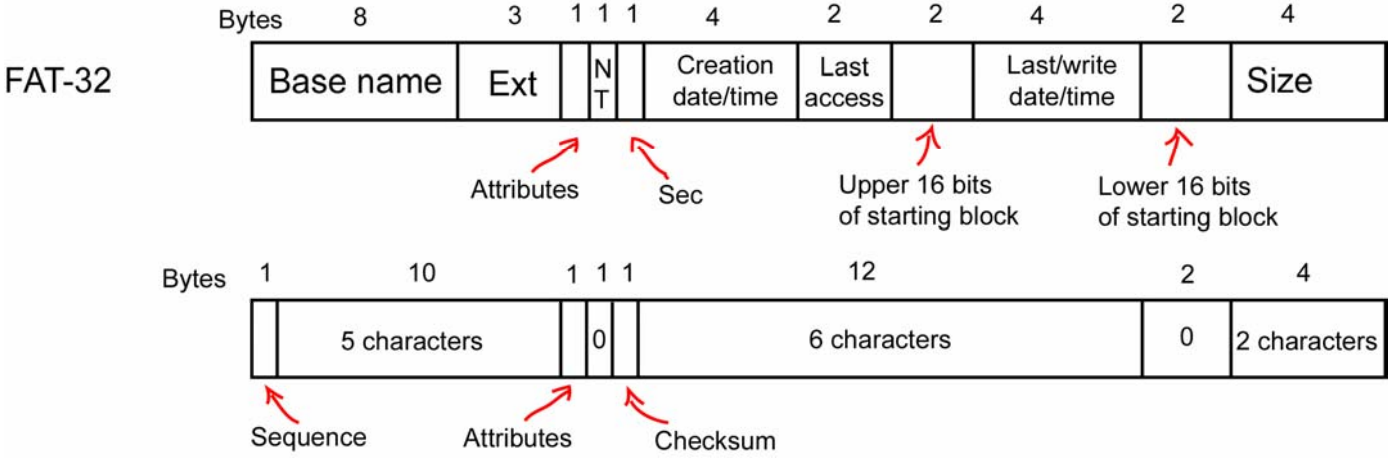
	Velikost diskové adresy	Velikost dat. bloku	Maximální velikost disk. oblasti
FAT-12	12 bitů	512 B – 8 KB	32 MB
FAT-16	16 bitů	512 B – 64 KB	4 GB
FAT-32	28 bitů	4 KB – 32 KB	32 GB (teoreticky 8TB)

Příklad: MS-DOS (2)

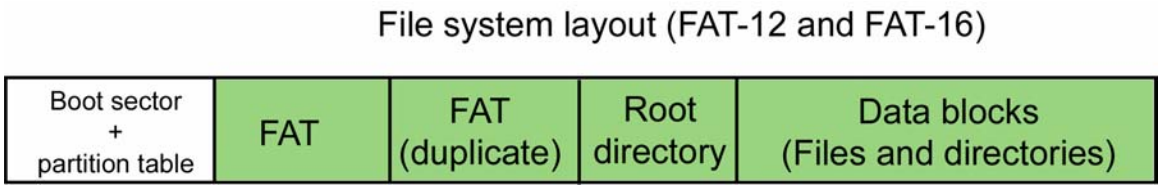
- **Položka adresáře ve FAT-12 a FAT-16** (32 Bytes)



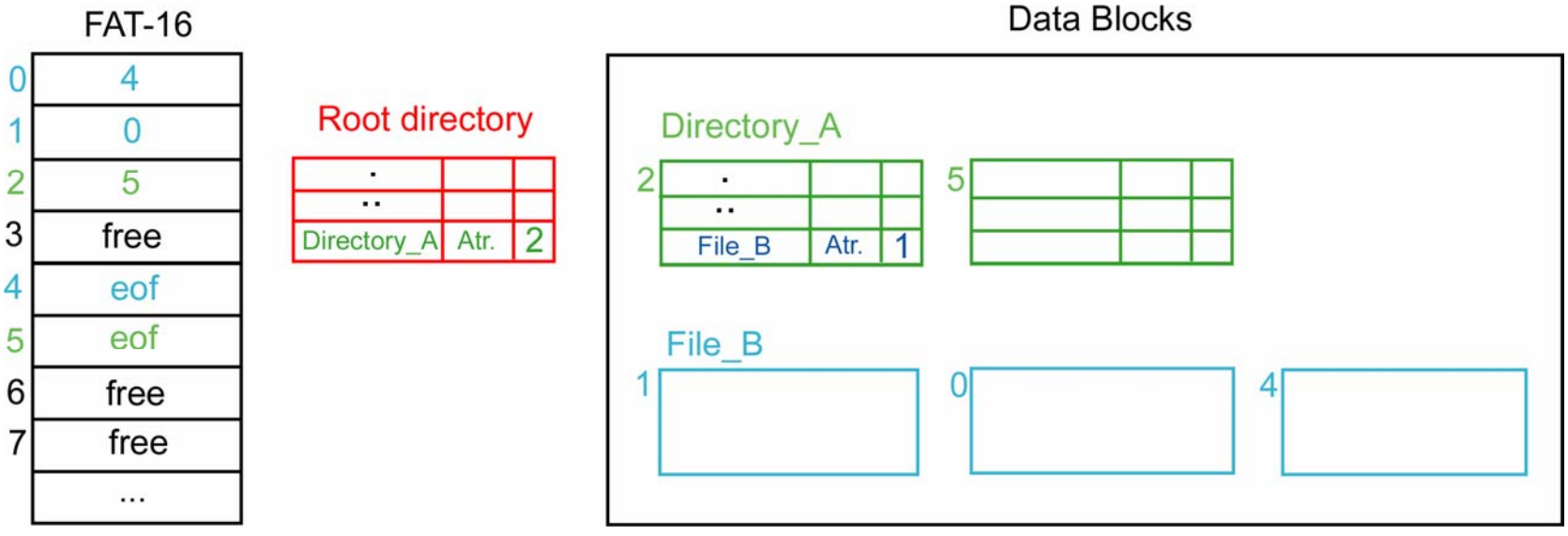
- **Položka adresáře ve FAT-32** (32 Bytes)
  - Každý soubor může mít dvě jména (jméno 8+3 a dlouhé jméno).
  - Dlouhé jméno je uloženo ve více položkách adresáře.



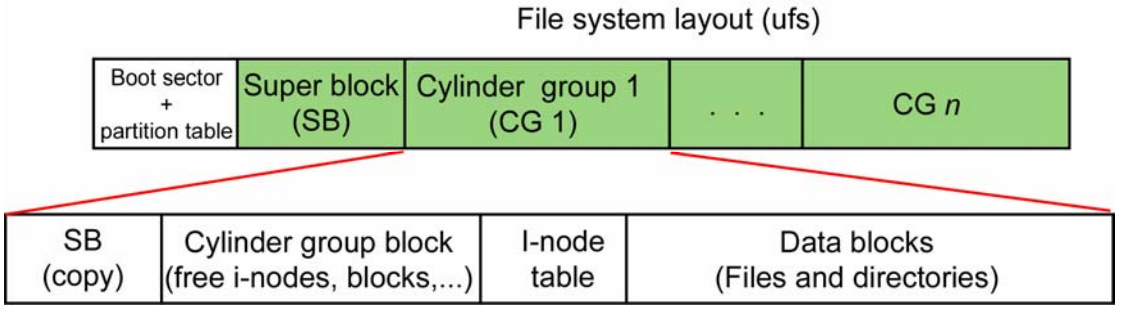
Příklad: MS-DOS (3)



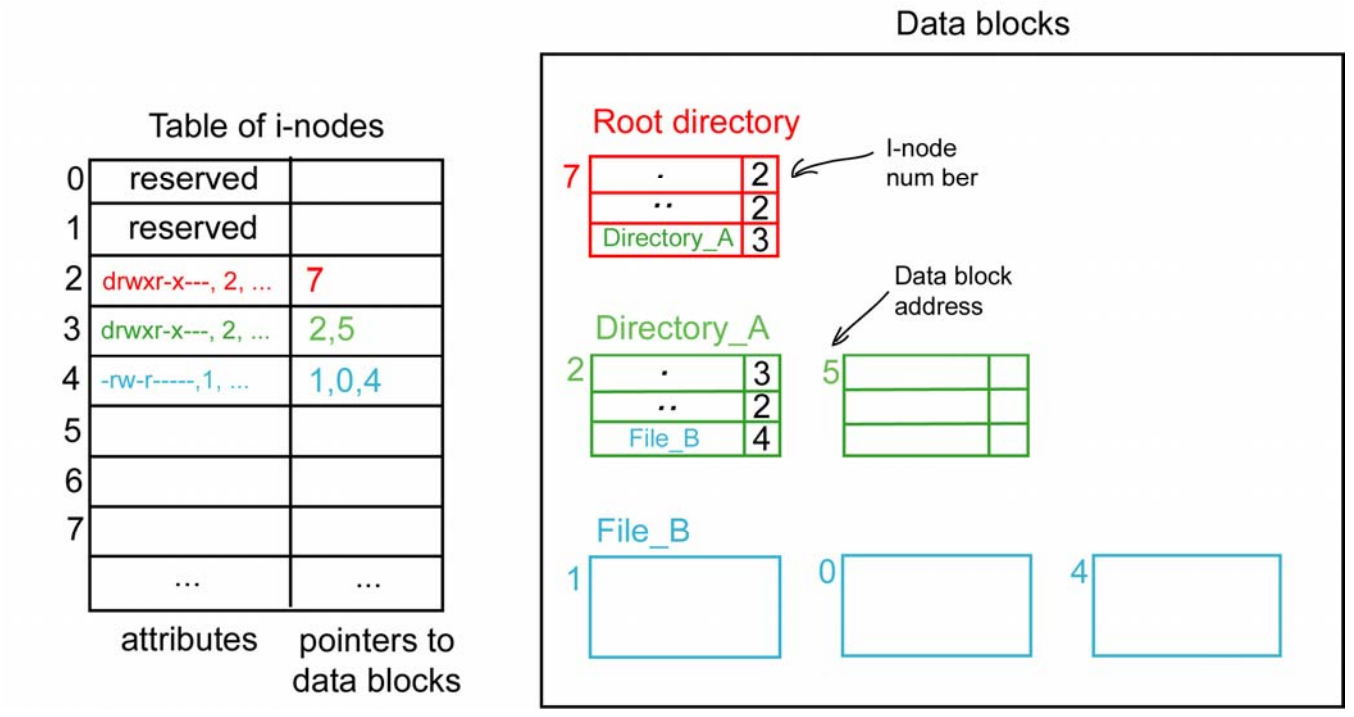
- **Přístup k souboru:** /Directory_A/File_B



Příklad: UNIX



- **Přístup k souboru:** /Directory_A/File_B



Příklad: NTFS

- **Jméno souboru**

  cesta (32767 znaků) + jméno (255 znaků) v UNICODE.
- **Podporuje**
  - přístupová práva na úrovni jednotlivých uživatelů
  - detailní přístupová oprávnění (např. přebírat vlastnictví, měnit oprávnění, ...)
  - hard linky i symbolické linky
  - alokaci po datových streamech (max. velikost 2^{64} B)
  - žurnálování
  - kompresi a kryptování dat

	Velikost diskové adresy	Velikost dat. bloku	Maximální velikost disk. oblasti
NTFS	64 bitů (prakticky 32)	512 B – 64 KB	256 TB (teoreticky 2^{64} bloků)

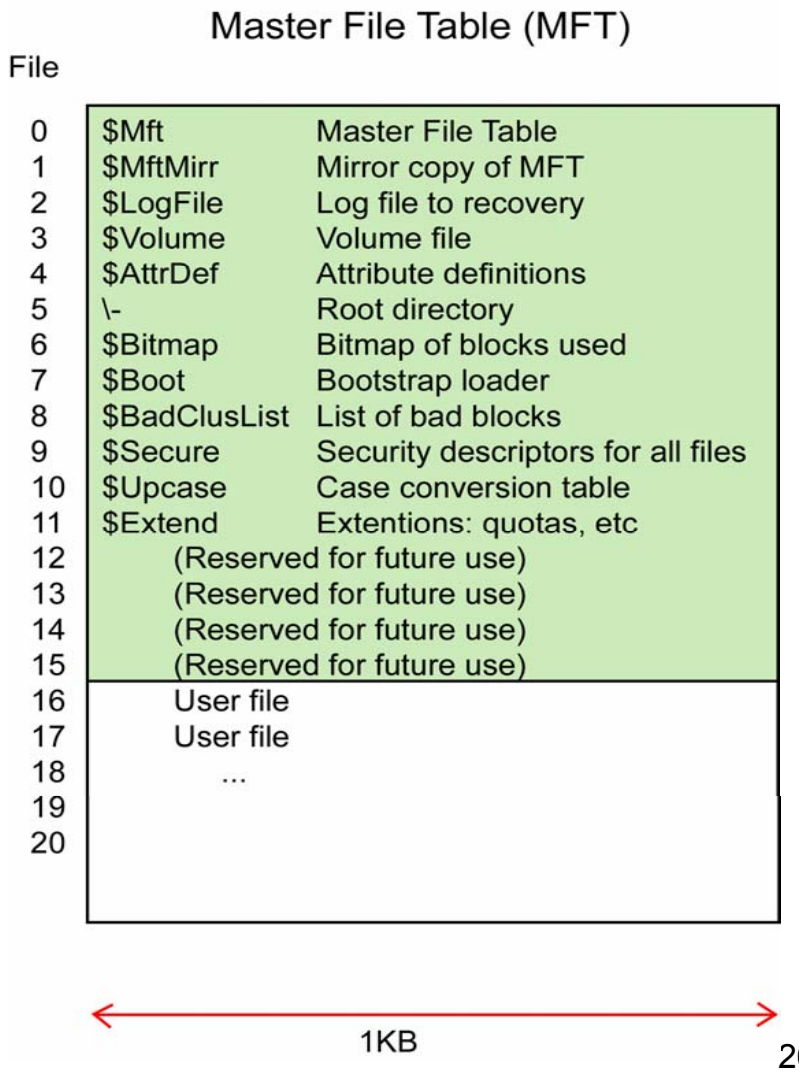
Příklad: NTFS (2)

NTFS volume layout

Příklad: NTFS (3)

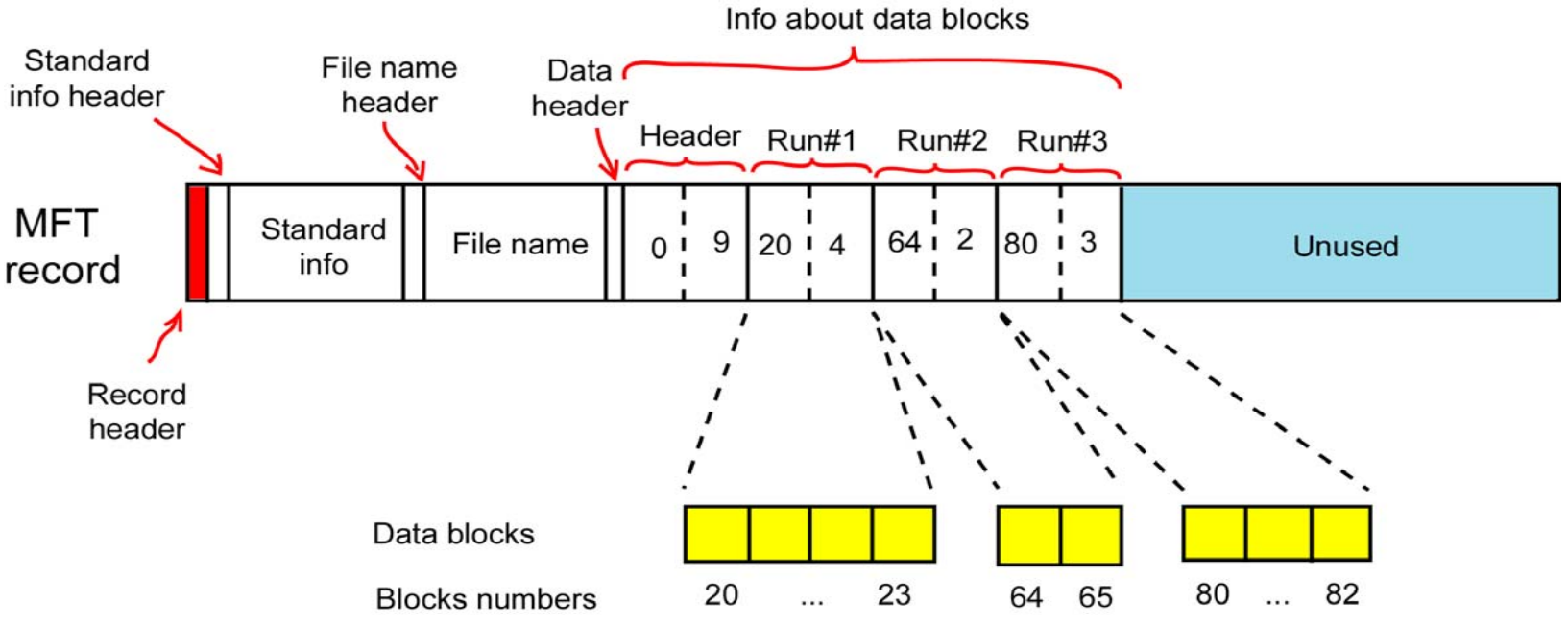
- **Master File Table (MFT)**
  - Položka v MFT popisuje jeden soubor/adresář.

- 1. položka – kopie MFT
- 2. položka – log soubor
  - záznam o všech změnách, kromě uživatelských dat
- 3. položka – informace o volume
- 5. položka – kořenový adresář



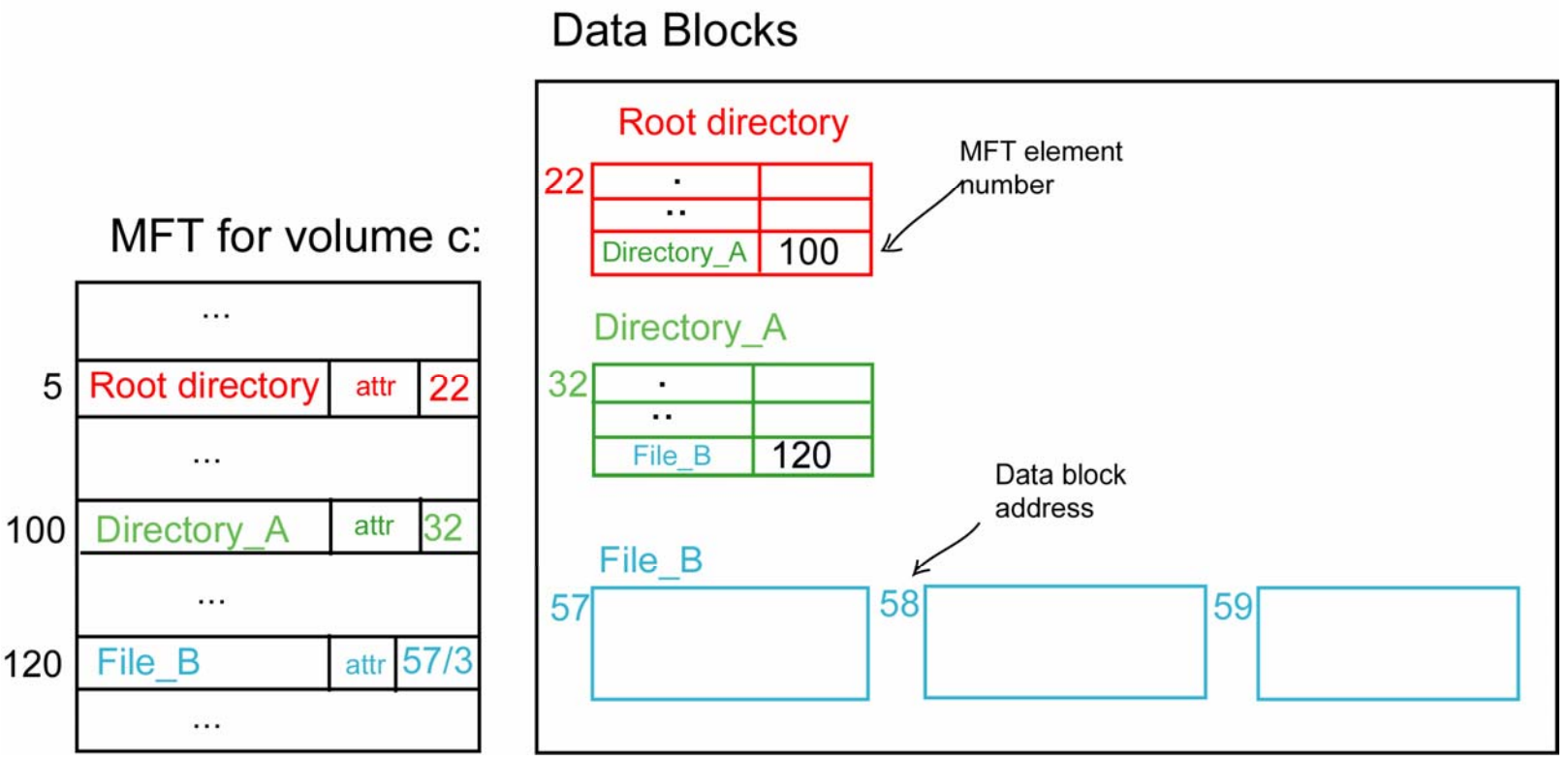
Příklad: NTFS (4)

- **Položka MFT**
  - se skládá z posloupnosti dvojic (attribute header a hodnota).
  - **rezidentní atribut** = attr. headrer i hodnota jsou v položce MFT.
  - **nerezidentní atributy** = attr. headrer je v položce MFT, ale hodnota je uložena v datových blocích.



Příklad: NTFS (5)

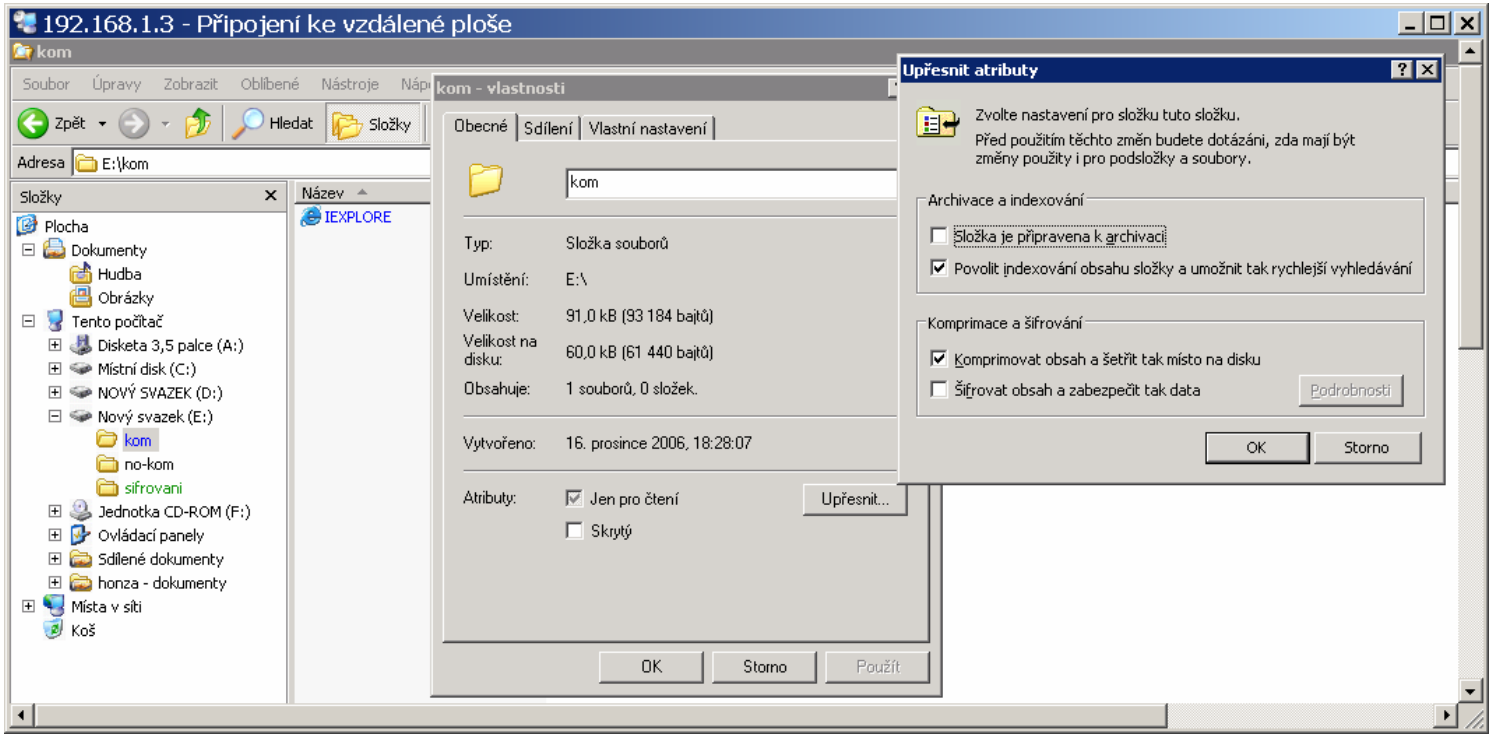
- **Přístup k souboru:** C:\Directory_A\File_B



Příklad: NTFS – komprese

- **Transparentní komprese** souborů, složek a volumů.
- Nelze komprimovat zašifrované soubory.
- Umožňuje pouze Windows XP Professional.

Průzkumník Windows → Vlastnosti → Obecné → Upřesnit → Komprimovat

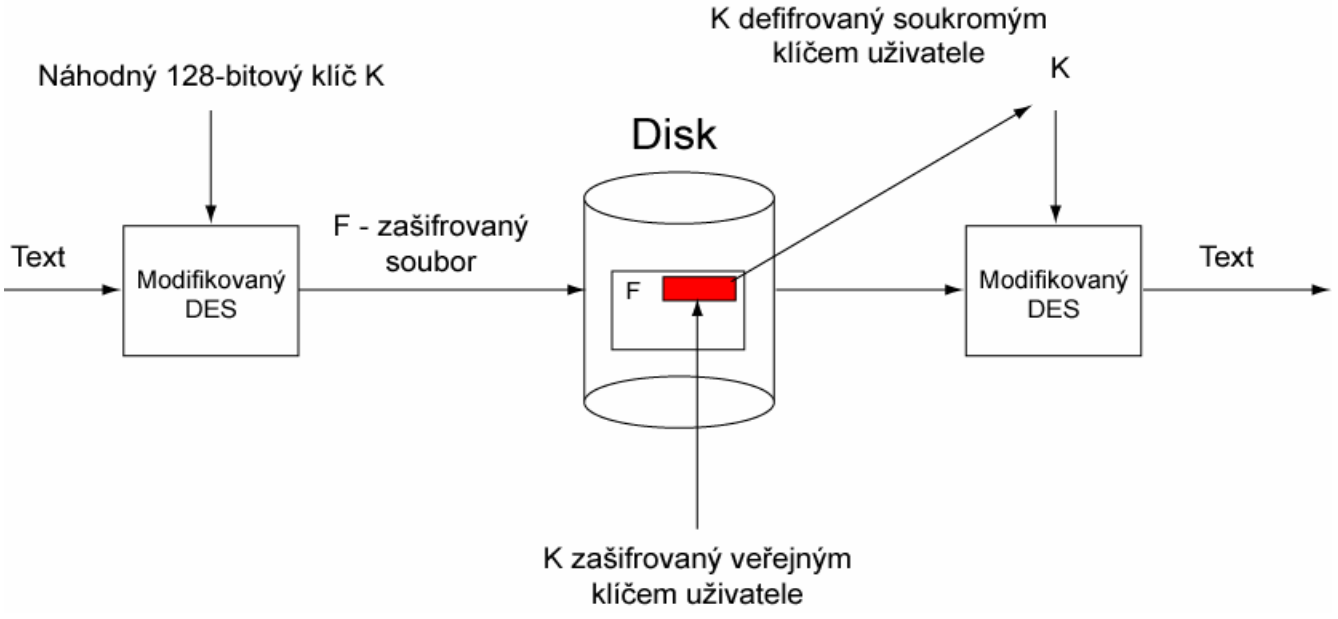


Příklad: NTFS – šifrování

- **Transparentní šifrování** souborů, složek a volumů.
- Umožňuje pouze Windows XP Professional.
- Nelze šifrovat
  - systémovou složku C:\Windows
  - komprimované soubory
- Přistupovat k zašifrovaným souborům může explicitně pouze jejich vlastník (lze povolit i dalším uživatelům) .

Příklad: NTFS – šifrování (2)

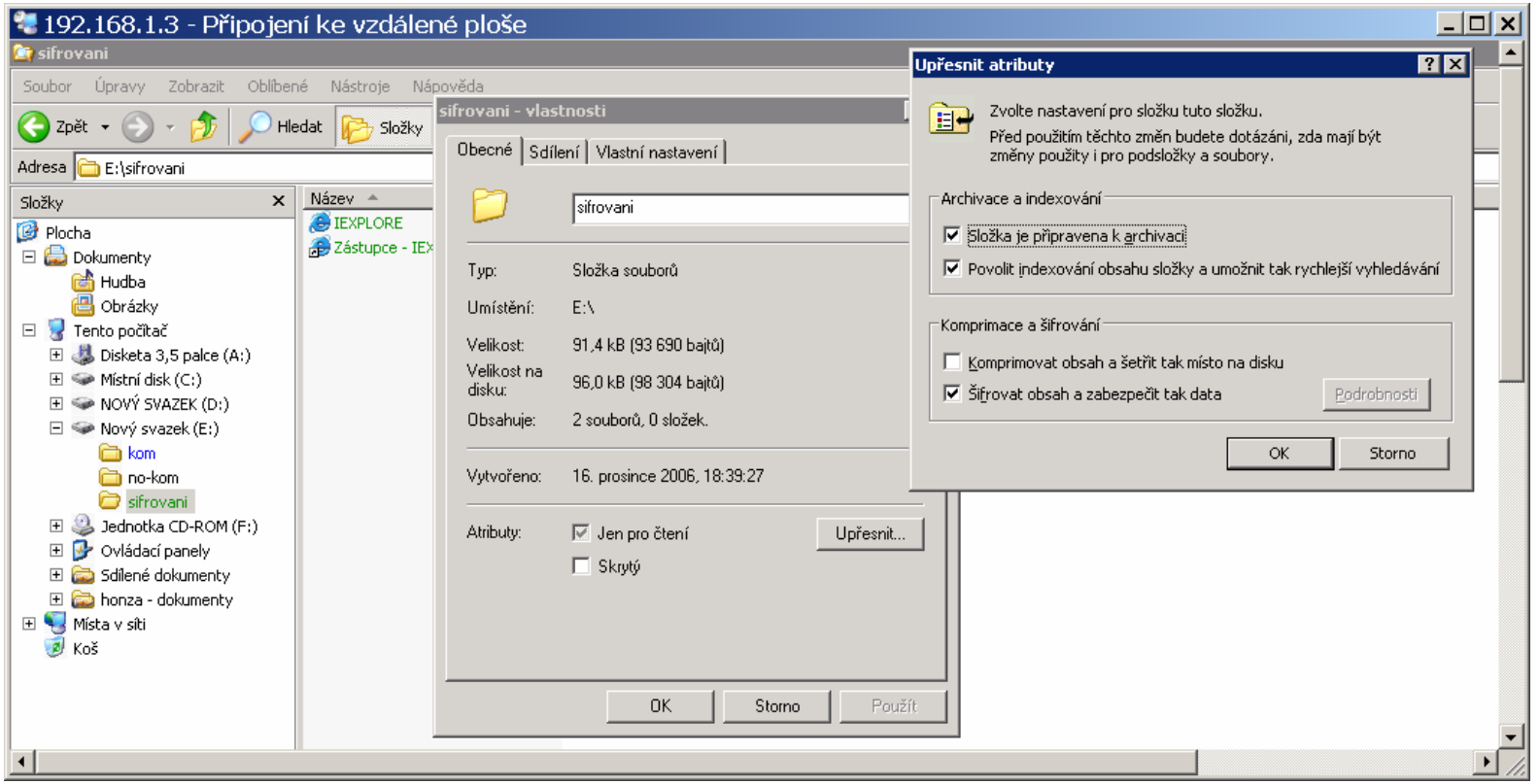
- Šifrování provádí ovladač **EFS (Encrypting File System).**
- Pro každý soubor je náhodně vygenerován symetrický klíč, kterým je zašifrován..
- Symetrický klíč je zašifrován veřejným klíčem uživatele a uložen na disku.
- Privátní klíč uživatele je zašifrován symetrickým algoritmem pomocí uživatelského hesla.



Příklad: NTFS – šifrování (3)

- **Pomocí GUI**

  `Průzkumník Windows → Vlastnosti → Obecné → Upřesnit → Šifrování`



Výkonnost FS

- Čtení dat. blok z disku je výrazně pomalejší než čtení slova z hlavní paměti.

- **Vyrovnávací paměť (block cache)**
  - Obsahuje některé informace načtené z FS na disku.
  - Část RAM.
  - Statická cache (fixní velikost, např. 10% RAM).
  - Dynamická cache (proměnná velikost, např. od 5 do50 % RAM).

- Pokud není místo ve vyrovnávací paměti, některé dat. bloky musí být odsunuty zpět na disk.

- Pro výběr dat. bloků, které odsuneme, používáme **modifikované algoritmy pro náhradu stránek** (FIFO, LRU, second chance).

Výkonnost FS (2)

- Strategie zápisu modifikovaných dat. bloků:
  - **Write through cache** (MS-DOS)
    - Všechny změny ve FS se zapisují okamžitě jak do vyrovnávací paměti tak do FS na disku (bezpečné, ale málo efektivní).
  - **Nonwrite through cache** (UNIX)
    - Čtení z vyrovnávací paměti je synchronní (data jsou současně načtena z FS na disku do vyrovnávací paměti i do procesu).
    - Zápis obsahu vyrovnávací paměti do FS na disku je asynchronní – prováděný periodicky pomocí systém. démona (např. **syncer** v UNIXu každých 30 sekund).

Žurnálované FS

- **Metadata**
  - důležité dat. struktury FS (super block, i-nodes, adresáře, ...).
- **Problém s tradičními FS**
  - Pokud při zápisu metadat na disk dojde k výpadku systému, FS může být v nekonzistentním stavu.
- **Řešení**
  - **Kontrolovat celý FS** po restartu systému (např. pomocí *scandisk* v MS Windows nebo *fsck* v UNIXu).
  - **Používat žurnálování u FS**.

Žurnálované FS (2)

- **Princip**
  - Ve FS je speciální soubor "**journal**", který obsahuje informace o transakcích na disku.
  - Při provádění disk. operace se nejdříve zapíše na disk informace o transakci, a potom se teprve operace začne provádět.

  - **Při výpadku FS**:
    - buď byla operace korektně dokončená a FS je konzistentní,
    - nebo se operace nedokončila, pak na základě „jurnalu" opravíme stav FS do konzistentního stavu (dokončíme nebo zrušíme operaci).
- **FS podporující jurnálování**
  - ntfs (MS Windows), ext3 (Linux), ufs (Solaris), jfs (AIX),...

Operační systémy

Přednáška 12: Vstup/výstup

Hlavní funkce V/V

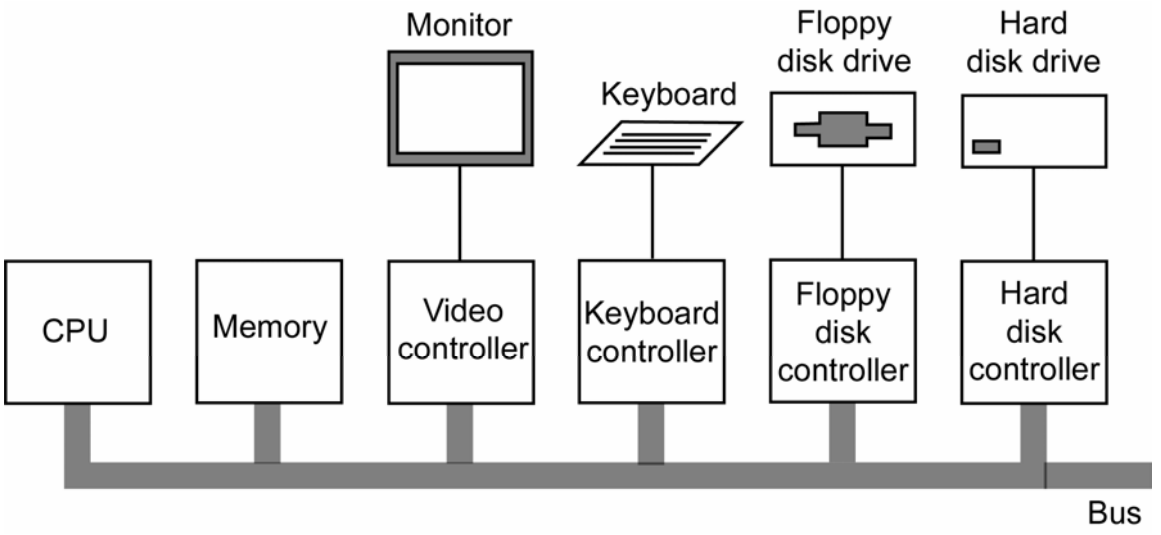
- OS musí **spravovat všechny V/V zařízení počítače**.
  - vyslat příkaz pro dané zařízení, reagovat na přerušení, reagovat na chyby.
  - zakázat neautorizovaný přístup.

- OS musí **poskytovat rozhraní** mezi V/V zařízeními a zbytkem systému.
  - jednotný přístup k zařízením (na úrovni API),
  - přístup k zařízením na vyšší úrovni abstrakce,
  - virtualizace zařízení (spooling),
  - jednotné pojmenování,
  - jednotné reakce na chyby.

Typy V/V zařízení

- **Blokově orientované**
  - Informace je uložená ve stejně velkých blocích, každý je přímo adresovatelný.
  - Lze číst/zapisovat každý blok nezávisle od ostatních bloků.
  - Například: disky, diskety,…
- **Znakově orientované**
  - Informace je uložena jako posloupnost bytů.
  - Není možné přistupovat přímo k jednotlivým znakům.
  - Například: tiskárny, síťová rozhraní, myši, pásky,...
- **Některá zařízení nelze zařadit do těchto skupin**
  - (např. časovače, …).

Periferní zařízení



- V/V zařízení se obecně skládají z dvou částí:
  - **řadič zařízení**,
  - **samotné V/V zařízení**.
- Komunikace se samotným V/V zařízením probíhá na **nízké úrovni**.
- Úkolem řadičem je poskytnout **jednoduchého rozhraní** pro OS.

Periferní zařízení (2)

- Každý řadič má:
  - **řídící registry**
    - Pomocí zápisu do registrů, CPU může udílet příkazy V/V zařízení
    - Pomocí čtení těchto registrů, CPU může získat informaci o stavu V/V zařízení
  - **vyrovnávací paměti** (ne vždy)
    - CPU může číst/zapisovat do těchto pamětí.

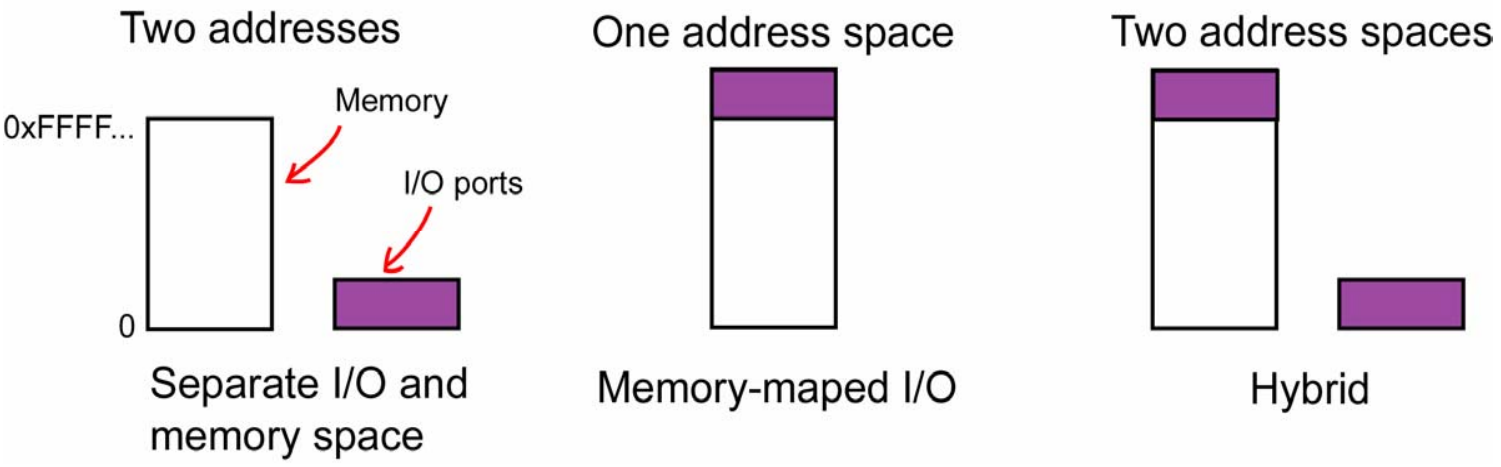
- **Jak CPU komunikuje s řídícími registry a vyrovnávacími paměťmi ?**
  - Oddělený V/V prostor a paměť.
  - V/V prostor mapovaný do paměti.

Oddělený V/V prostor a paměť

- Adresový prostor paměti je jiný než V/V prostor.
- Každý řídící registr má přiřazeno **číslo V/V portu** (8- or 16-bit integer).
- **CPU používá speciální V/V instrukce**:
  - **IN REG, PORT**
    - pro zkopírování hodnoty řídícího registru PORT do registru REG v CPU
  - **OUT PORT,REG**
    - pro zápis hodnotu z REG do řídícího registru PORT.
- Následující instrukce jsou rozdílné!!!

  IN R0, 4 x **MOV R0,4**

V/V prostor mapovaný do paměti

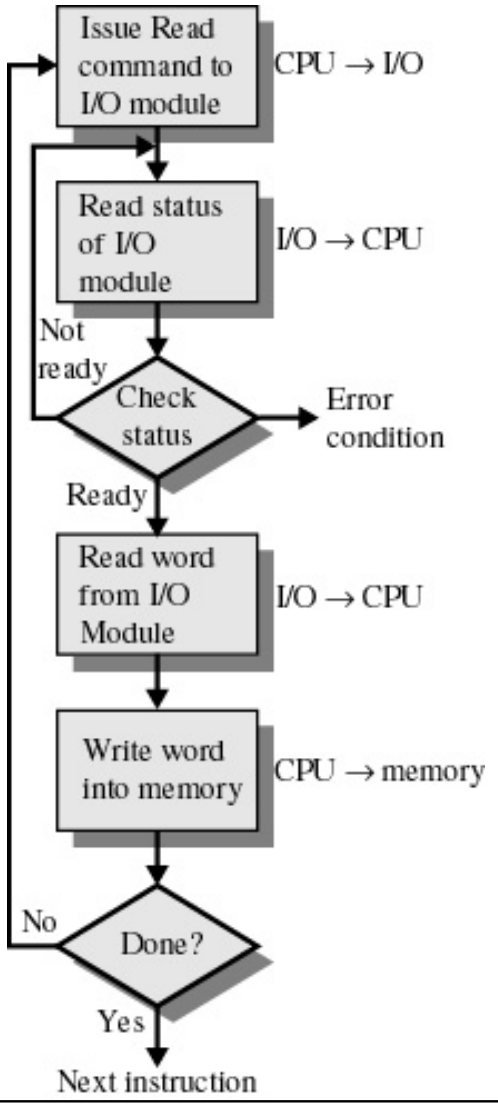


- Ve **V/V prostoru mapovaném do paměti**, každý řídící registr má přiřazenu jedinečnou paměťovou adresu.

- V **hybridním modelu**, jsou vyrovnávací paměti mapovány do paměti, ale řídící registry jsou odděleny pomocí čísel portů.

Programovatelný V/V

- CPU se stará o celou V/V operaci.
- CPU zahájí V/V operaci.
- Potom CPU čeká na její dokončení pomocí aktivního čekání.



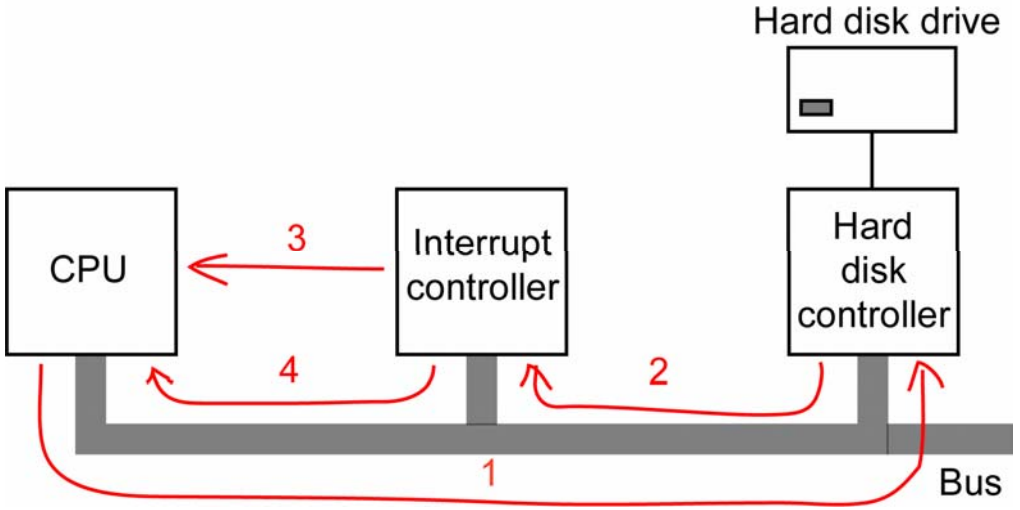
Programovatelný V/V (2)

Příklad: Uživatelský proces chce vytisknout osmi znakový řetězec "ABCDEFGH" na tiskárnu. Předpokládáme V/V prostor mapovaný do paměti a tiskárnu bez vyrovnávací paměti.

- Uživatelský proces použije systémové volání.
- OS provádí následující kód:

```
copy_from_user(buffer, p, count);          /* p is the kernel buffer*/

for ( i=0; i<count; i++ ) {                /* loop on every character */
   while (*printer_status_reg != READY);   /* loop until ready */
   *printer_data_register = p[i];          /* output one character */
}

return_to_user();
```

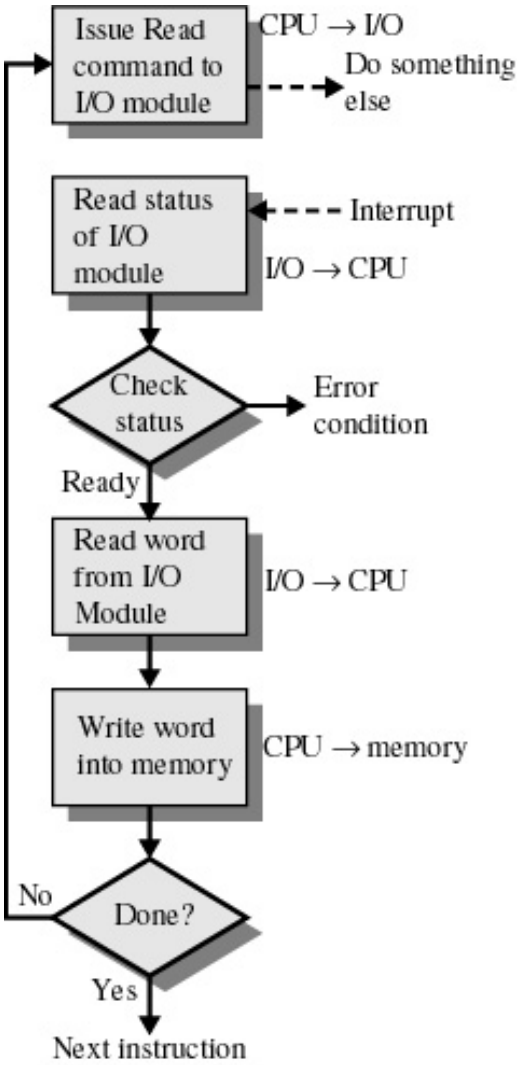
Přerušení



- **V/V operace pomocí přerušení**
  1. **CPU řekne řadiči** co má dělat tím, že zapíše příkaz do řídícího registru.
  2. Když řadič V/V zařízení dokončí operaci, **pošle signál do řadiče přerušení**.
  3. Pokud je řadič přerušení připraven přijmout přerušení, **informuje o tom CPU**.
  4. Řadič přerušení **pošle přes sběrnici číslo V/V zařízení do CPU**.

V/V používající přerušení

- CPU je zahájí V/V operaci a pak pokračuje v jiné práci.
- Není zde aktivní čekání.
- Když jsou data připravena je vyvoláno přerušení a CPU zkopíruje data do paměti.



V/V používající přerušení (2)

Příklad: Uživatelský proces chce vytisknout osmi znakový řetězec “ABCDEFGH” na tiskárnu. Předpokládáme V/V prostor mapovaný do paměti a tiskárnu bez vyrovnávací paměti.

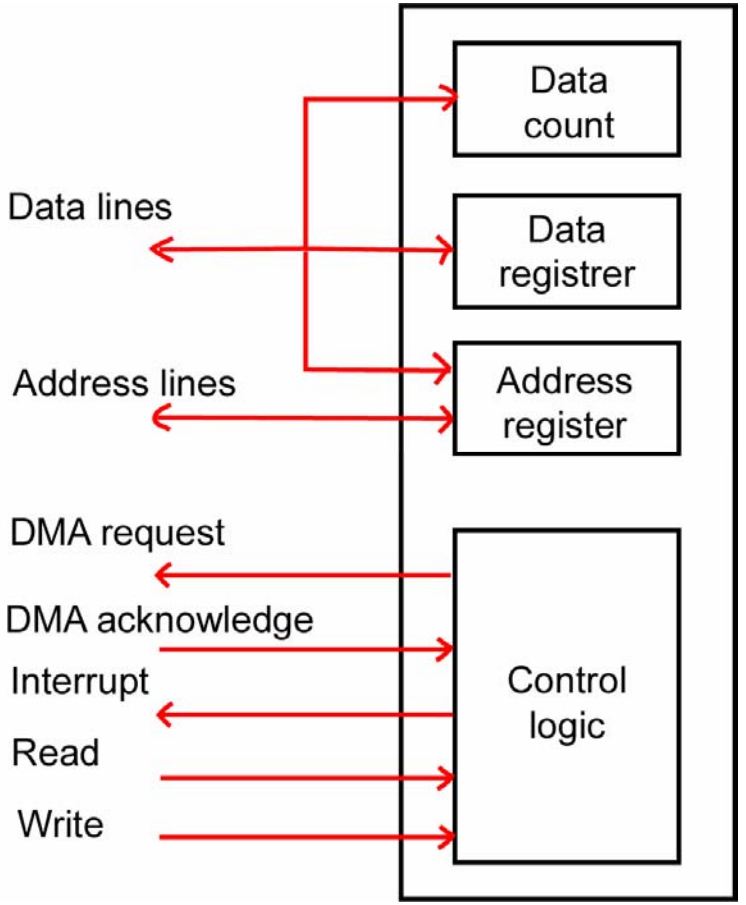
Uživatelský proces použije systémové volání.
OS provádí následující kód:

```
/* Code executed when                */
/* the print system call is made  */

copy_from_user(buffer, p, count);
enable_interrupts();
while (*printer_status_reg != READY);
*printer_data_register = p[0];
scheduler();
```

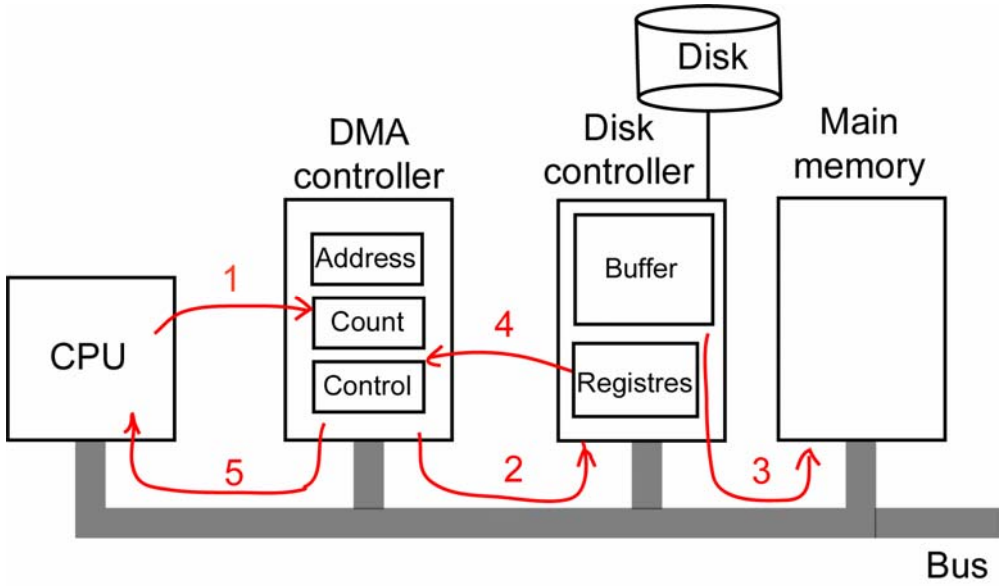
```
/* Interrupt service  */
/*     procedure       */

i = 1;
if (count == 0) {
  unblock_user();
} else {
  *printer_data_register = p[i];
  count = count – 1;
  i = i + 1;
}
acknowledge_interrupt();
return_from_interrupt();
```

Direct Memory Access (DMA)

- **Speciální DMA chip** může řídit tok dat mezi pamětí a V/V zařízením bez asistence CPU.



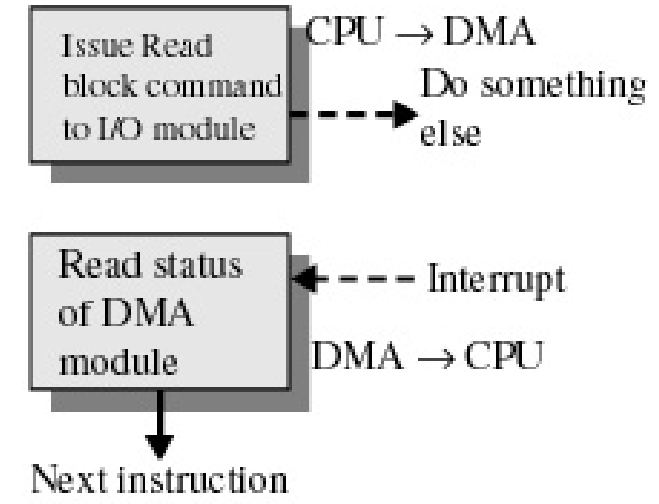
DMA (2)



- **V/V operace pomocí DMA**
  1. CPU naprogramuje DMA řadič nastavením jeho registrů.
  2. DMA pošle příkaz do řadiče V/V zařízení.
  3. Přenos dat do paměti.
  4. Když je přenos dat dokončen, řadič disku pošle potvrzení do řadiče DMA.
     - DMA inkrementuje adresu paměti a dekrementuje čítač dat.
     - Pokud je čítač větší než 0, opakují se kroky 2-4.
  5. Pokud je čítač roven 0, potom řadič DMA přeruší CPU.

V/V používající DMA

- Přesouvá data přímo do/z paměti.
- K přerušení dojde až když je V/V operace dokončena.
- CPU pouze zahájí a dokončí V/V operaci.



(c) Direct memory access

V/V používající DMA (2)

Příklad: Uživatelský proces chce vytisknout osmi znakový řetězec "ABCDEFGH" na tiskárnu. Předpokládáme V/V prostor mapovaný do paměti a tiskárnu bez vyrovnávací paměti.

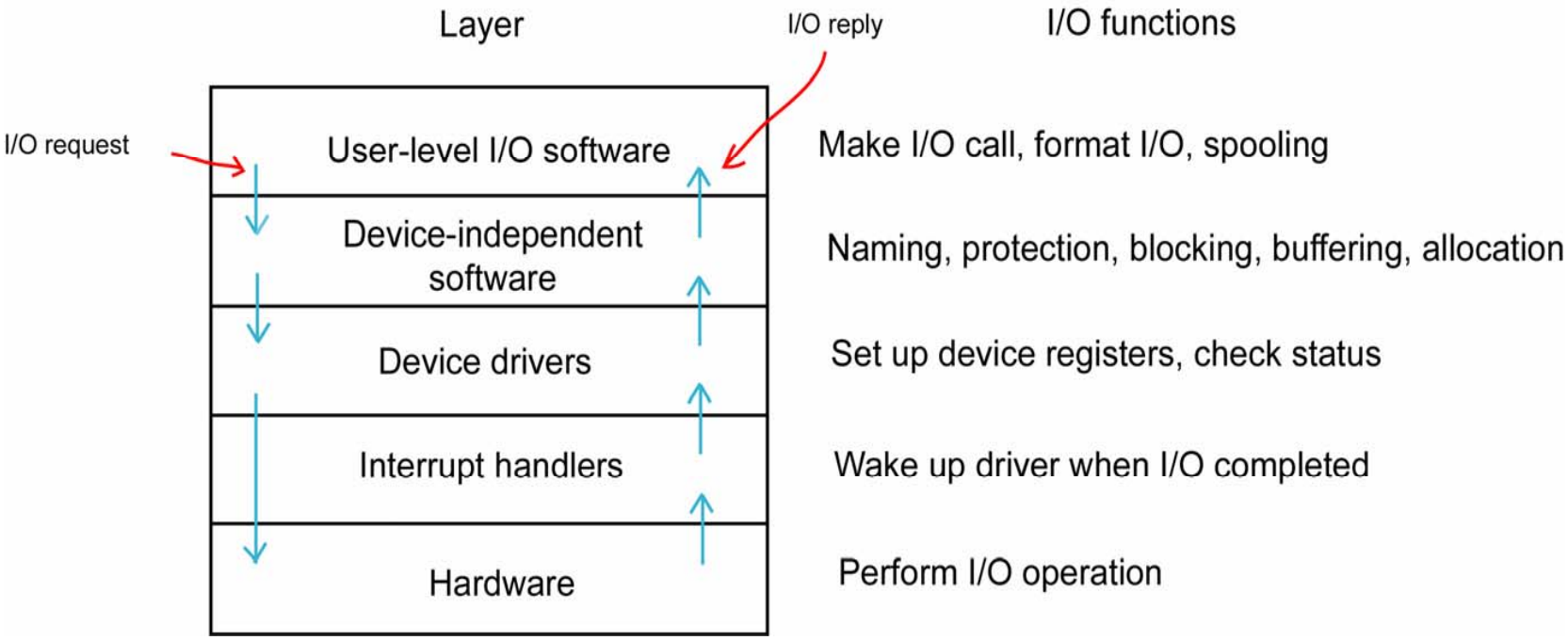
Uživatelský proces použije systémové volání.
OS provádí následující kód:

```
/* Code executed when          */
/* the print system call is made */

copy_from_user(buffer, p, count);
set_up_DMA_controller();
scheduler();
```

```
/* Interrupt service  */
/*     procedure      */

acknowledge_interrupt();
unblock_user();
return_from_interrupt();
```

Struktura V/V softwaru

- Typicky organizován do čtyř vrstev.
- Každá vrstva má **přesně definovanou funkci**, kterou plní, a **přesně definované rozhraní** k sousedním vrstvám.
- Funkce a rozhraní je **závislé na systému**.



User-Space I/O Software

- Většina V/V software je součástí OS.

- Malá část je v **knihovnách**, které jsou spojené s uživatelským programem a běží mimo jádro.
  - **knihovní funkce**

    printf("The square of %3d is %6d\n", i, i*i);
  - **systémová volání**

    count=write(fd, buffer, nbytes);

- **Spooling system** = cesta umožnit přístup k jednotlivým V/V zařízením ve více úlohovém systému (e.g. printer, file transfer over a network,…).

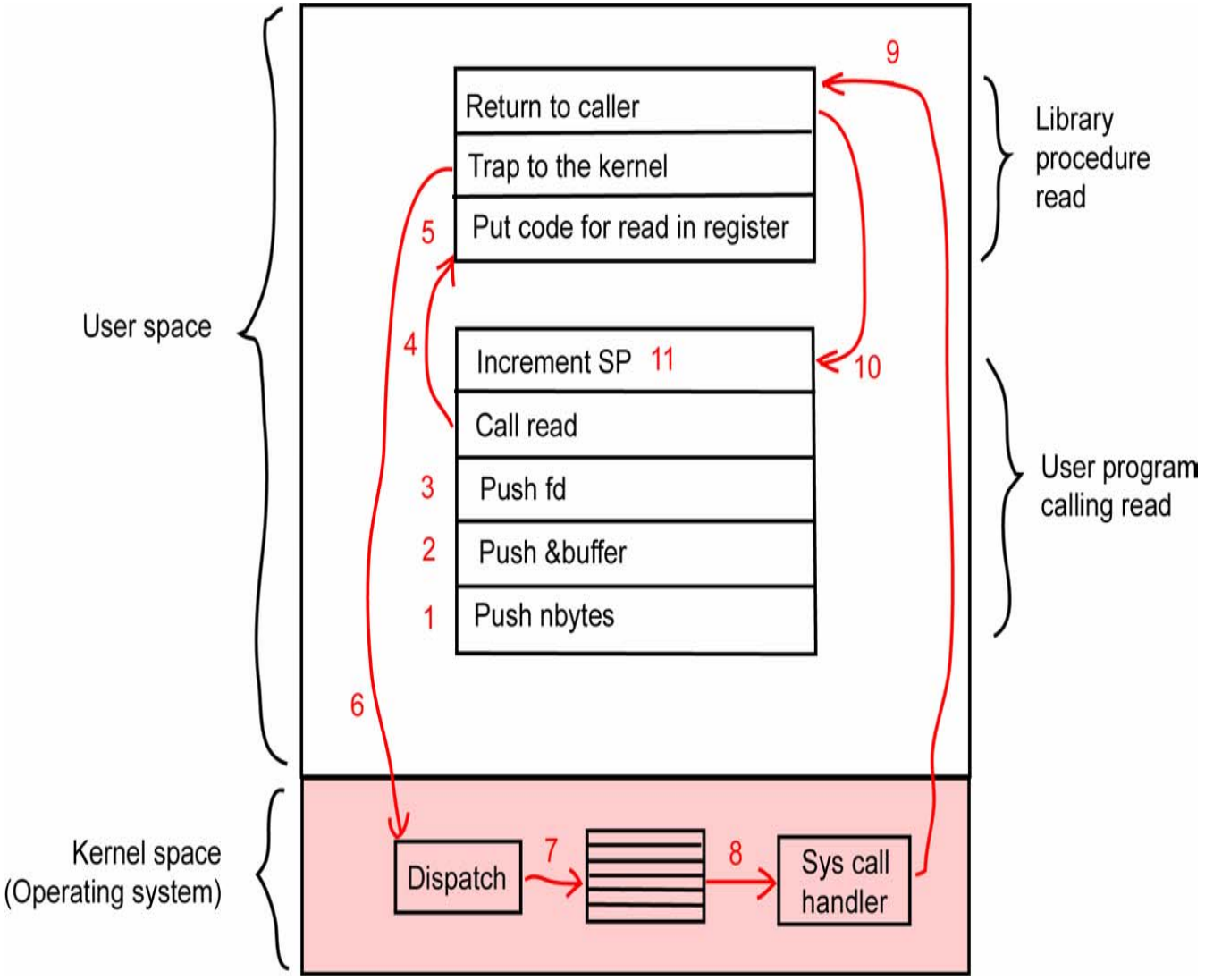
Volání služeb jádra

- Nástroj **rozhraní mezi aplikacemi a jádrem OS.**

- **Volání jsou přímo dostupná na úrovni**
  - symbolického strojového jazyka (assembleru),
  - vyššího programovacího jazyka (C/C++).

- **Metody předávání parametrů** mezi aplikací a jádrem
  - v **registrech** (jsou dostupné aplikaci i jádru),
  - v **tabulce** uložené v hlavní paměti a adresa tabulky se umístí do
    - registru,
    - na zásobník (je dostupný aplikaci i jádru).

- Příklad: count = read(fd, &buffer, nbytes)



Volání služeb jádra (2)

- **Kroky systémového volání:**

- 1-3 aplikace umístí parametry na zásobník,

4 aplikace zavolá knihovní funkci,

5 **každé systémové volání má přiřazeno unikátní číslo**, knih. funkce vloží toto číslo do příslušného registru,

6 knih. funkce provede instrukci TRAP, CPU se přepne do kernel modu

7 jádro zkontroluje číslo systémového volání a pak zavolá odpovídající „system call handler“,

8 běží „system call handler“,

9 CPU se přepne do uživatelského režimu a řízení se vrací knih. funkci,

10 knih. funkce vrací řízení aplikaci,

11 aplikace vyčistí zásobník a pokračuje.

Device-Independent I/O Software

- **Hlavní funkce**
  - **Jednotné rozhraní pro ovladače**
    - Jednoduché přidávání nových ovladačů.
    - Mapování symbolických jmen zařízení na konkrétní ovladače

      (např. *major* a *minor* čísla v UNIXu).
  - **Buffering**
    - Vyrovnávací paměti v uživatelském prostoru, v prostoru jádra.
  - **Error reporting**
  - **Alokace a uvolňování jednotlivých V/V zařízení**
  - **Velikost datových bloků nezávislých na jednotlivých zařízení**

Device Drivers

- **Každé V/V zařízení** připojené k počítači **potřebuje** nějaký device-specific kód (**device driver**), který ho umí řídit.

- **Hlavní funkce**
  - **Překlad „device independent“ požadavků** pro konkrétní zařízení
    (čtení datového bloku -> určení cylindru, hlavičky, sektoru)
  - **Komunikace s řadičem V/V zařízení**.
    (zapsání příkazu do řídícího registru)
  - **Řazení více požadavků do fronty.**
  - **Zablokování a čekání na dokončení V/V operace**
  - **Error handling**

Interrupt Handlers

- **Po HW přerušení, SW musí provést**:
  1. **Uložení některých registrů**, které nebyly uloženy HW.
  2. **Nastavení kontextu** přerušovací rutiny (ISP), např. nastavení TLB, MMU, tabulky stránek,…
  3. **Nastavení zásobníku** pro ISP.
  4. **Potvrdit přerušení řadiči přerušení**.
  5. **Kopírovat registry** přerušeného procesu z dočasného umístění do prostoru procesu.
  6. **Spustit ISP**.
  7. Vybrat další proces, který poběží jako další.
  8. **Nastavit kontext pro další proces**.
  9. **Nahrát registry nového procesu.**
  10. **Spustit další proces**.

Operační systémy

Přednáška 13: Secure shell (ssh)

Secure shell – SSH

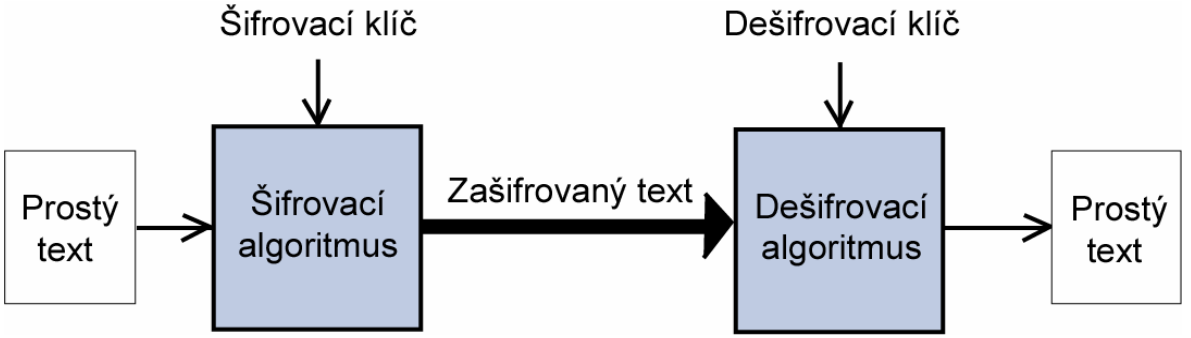
- **Softwarové řešení síťového zabezpečení** (na aplikační úrovni).

- **Založené na architektuře klient/server**
  - **klienti**: **`ssh`**, **`slogin`**, **`scp`** (v Unixu)
    např. **`putty`**, **`winscp`** (MS Windows)
  - **server**: **`sshd`**
  - transportní protokol je TCP/IP a server obvykle naslouchá na portu 22
  - různé implementace SSH1, SSH2, OpenSSH, ...

- **Přehled vlastností:**
  - **Soukromí (šifrování)** = ochrana dat před rozkrytím (odposloucháváním)
  - **Integrita dat** = garance, že se data nezmění během přenosu
  - **Autentizace** = ověření identity (jak serveru tak uživatele)
  - **Autorizace** = definice co smí příchozí uživatel dělat
  - **Směrování** (tunelování) = zapouzdření jiného protokolu využívající služeb TCP/IP do šifrované relace SSH

Základy kryptografie I

- **Šifrování (šifra)**
  - proces kódování dat takovým způsobem, aby je nebylo možné přečíst neoprávněnými osobami



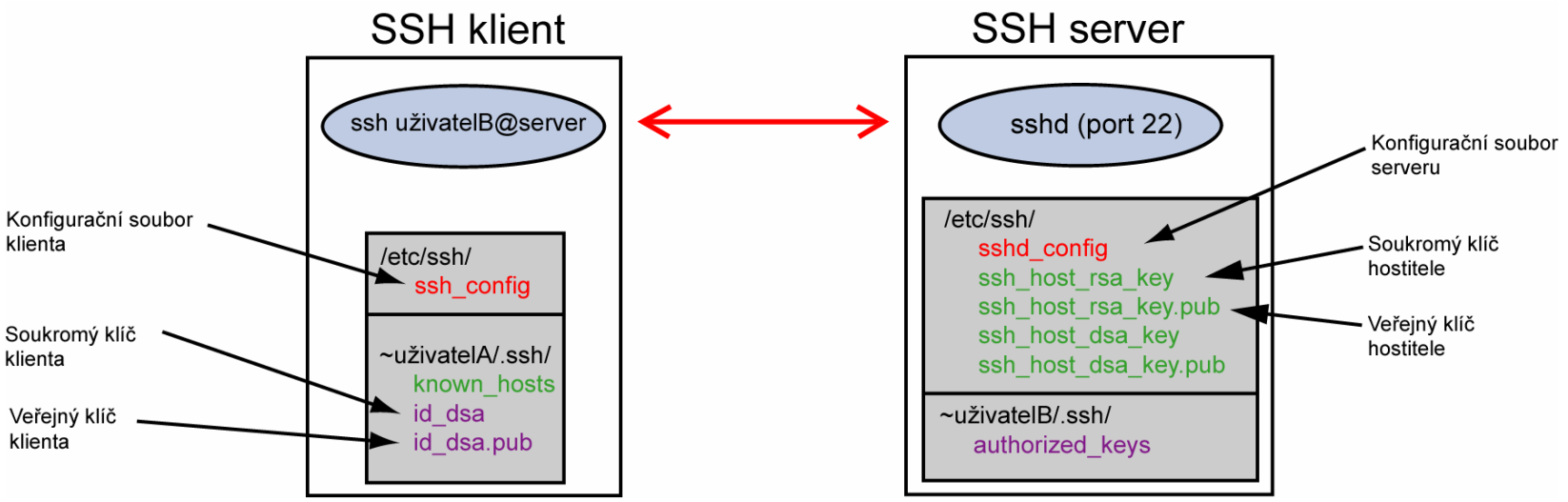
- **Šifrovací algoritmus**
  - konkrétní metoda, kterou se kódování provádí (např. DES, RSA, DSA, ...)
  - považuje se za **bezpečný** tehdy, když je pro ostatní osoby **“neproveditelné“ přečíst data bez znalosti klíče**
  - v současnosti **nelze dokázat, že nějaký algoritmus je 100% bezpečný**
- **Kryptoanalýza**
  - pokus dešifrování dat bez znalosti klíče

Základy kryptografie II

- **Symetrické šifry (šifry s tajným klíčem)**
  - k zašifrování a rozšifrování se používá tentýž klíč
  - **výhoda:** rychlost šifrování/dešifrování
  - **nevýhoda:** problém s distribucí klíče
  - např. Blowfish, DES, IDEA, RC4

- **Asymetrické šifry (šifry s veřejnými klíči)**
  - používá se dvojce klíčů: **veřejný a soukromý klíč**
  - data zašifrovaná veřejným klíčem lze rozšifrovat pouze jeho soukromým klíčem
  - **nelze odvodit soukromý klíč z veřejného**
  - **výhoda:** odpadá problém s distribucí klíče
  - **nevýhoda:** pomalé šifrování/dešifrování
  - např. RSA, DSA, ElGamal, Elliptic Curve, ...

Ustanovení zabezpečeného spojení



1. **Klient kontaktuje server** (port TCP na serveru obvykle 22)
2. Klient a server si vzájemně sdělí **jaké verze SSH podporují**.
3. **Server se identifikuje klientovi** a dodá mu parametry relace
   - veřejný klíč hostitele
   - seznam šifrovacích, kompresních a autentizačních metod, které server podporuje
4. **Klient odešle serveru tajný klíč relace** (zašifrovaný pomocí veřejného klíče hostitele).
5. **Obě strany zapnou šifrování a dokončí autentikaci serveru** (klient čeká na potvrzovací zprávu od serveru).

Příklady

- **Připojení na neznámý server (první připojení)**

```
ssh trdlicka@dray1.feld.cvut.cz
```

The authenticity of host 'dray1.feld.cvut.cz (147.32.192.154)' can't be established
RSA key fingerprint is d8:d4:05:fe:a7:b5:a1:42:6b:79:d4:58:3e:fe:44:1f.
Are you sure you want to continue connecting (yes/no)? yes

- Pro kontrolu, zda dva klíče jsou stejné, se používá **otisk klíče (fingerprint).**

- **Jak získat otisk klíče?**

```
$ ssh-keygen -l -f ssh_host_rsa_key.pub
```

1024 d8:d4:05:fe:a7:b5:a1:42:6b:79:d4:58:3e:fe:44:1f ssh_host_rsa_key.pub

Autentizace klienta

- Po ustanovení zabezpečeného spojení klient pokouší prokázat svou identitu (existuje několik metod):
- **Heslem**
- **Veřejným klíčem klienta**
  1. klient pošle svůj veřejný klíč serveru
     (např. ~uživatelA/.ssh/id_dsa.pub)
  2. server se pokusí najít záznam o klíči
     (např ~uživatelB/.ssh/authorized_keys)
  3. server vygeneruje náhodný řetězec, zašifruje ho veřejným klíčem klienta a pošle ho klientovi
  4. klient rozšifruje zašifrovaný řetězec svým soukromým klíčem a pošle serveru

Příklady

- **Vygenerování dvojce klíčů**

  `ssh-keygen -t dsa`

- Klíče se explicitně uloží do souborů

  `~uživatelA/.ssh/id_dsa` (soukromý klíč)

  `~uživatelA/.ssh/id_dsa.pub` (veřejný klíč)

- Pokud zadáte přístupovou frázi, pak bude klíč zašifrován a při autorizaci pomocí klíče budete muset zadávat přístupovou frázi.

- **Přidání veřejného klíče klienta do souboru na serveru**

  `~uživatelB/.ssh/authorized_keys`

- **Autorizace pomocí veřejného klíče**

  uživatelA@klient$ `ssh uživatelB@server`

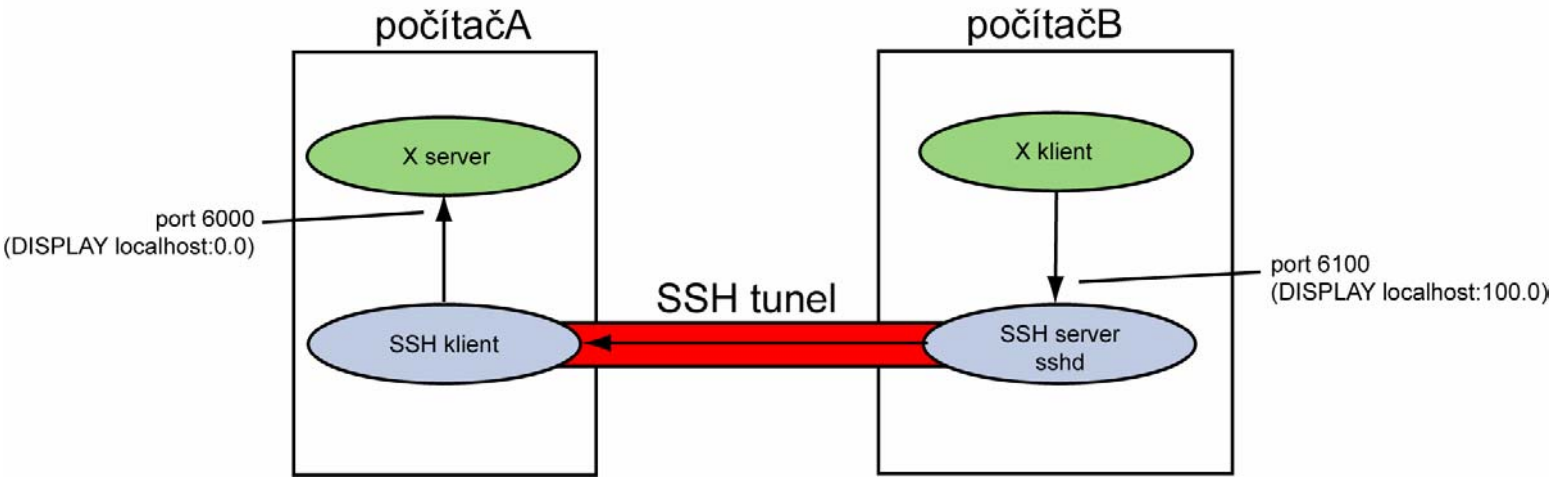
  uživatelB@server$

Příklady

- **Jak najít problém při připojování**

```
ssh -v uživatel@server
```

Tunelování protokolu X11



- Spojení přes **protokol X lze přesměrovat přes SSH**, které mu poskytne bezpečnost.
- Klient SSH si při připojení k serveru SSH vyžádá směrování protokolu X (musí být zapnuto na klientovi SSH a povoleno na serveru SSH):
  - **SSH server nastaví proměnnou**

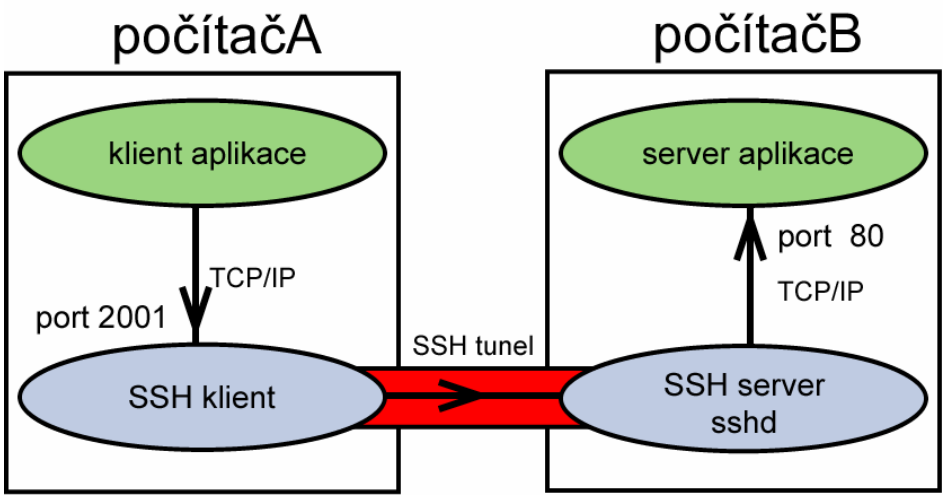
    `DISPLAY=localhost:pořadové_číslo_serveru.0`
  - SSH server začne **poslouchat na lokálním portu 6000+pořadové_číslo_serveru** a vše přeposílá na SSH klienta
  - klient SSH se chová jako X klient a obdržené výstupy posílá X serveru

Příklad

- **Připojení z počítačA na počítačB a zapnutí tunelování X11**

```
ssh -X uživatel@počítačB
```

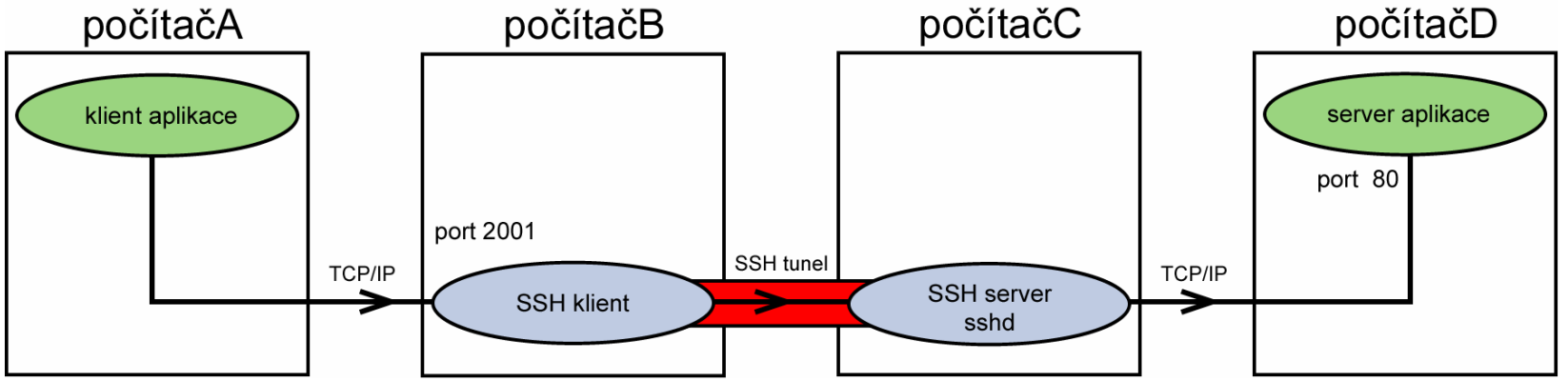
Tunelování portů (místní) I



- SSH klient na jedné straně přijímá požadavky na služby (TCP), zašifrované je posílá na SSH server, který je na druhé straně doručí cílovému příjemci.
- SSH tunel je pro aplikace **transparentní**.

```
ssh -L 2001:localhost:80 uživatel@počítačB
```

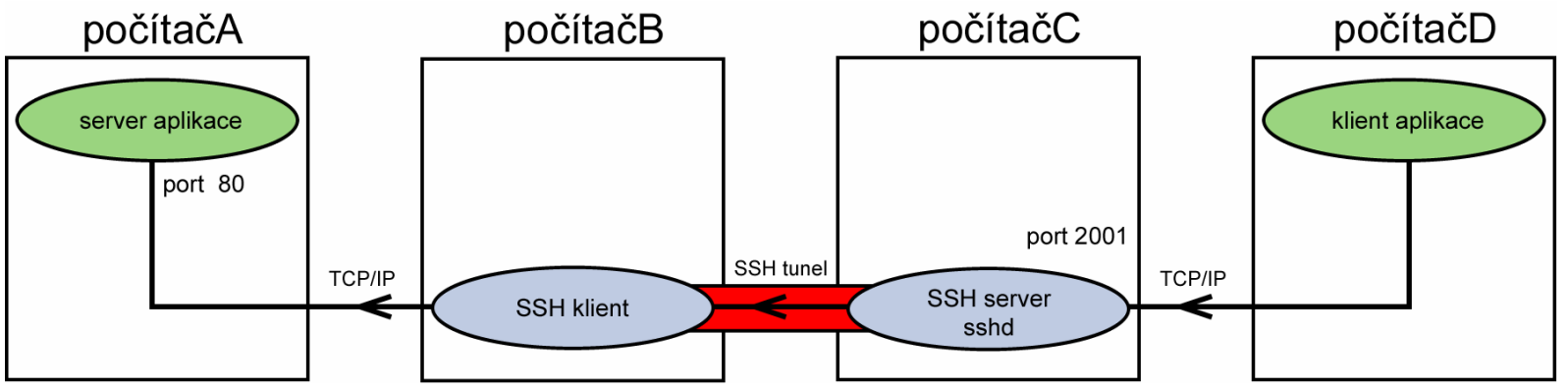
Tunelování portů (místní) II



- Jednotliví klienti a servery můžou běžet na různých počítačích.

```
ssh -g -L 2001:počítačD:80 uživatel@počítačC
```

Tunelování portů (vzdálené)




```
ssh -g -R 2001:počítačD:80 uživatel@počítačC
```