

Operační systémy

Úvod a definice pojmu.

Jan Trdlička



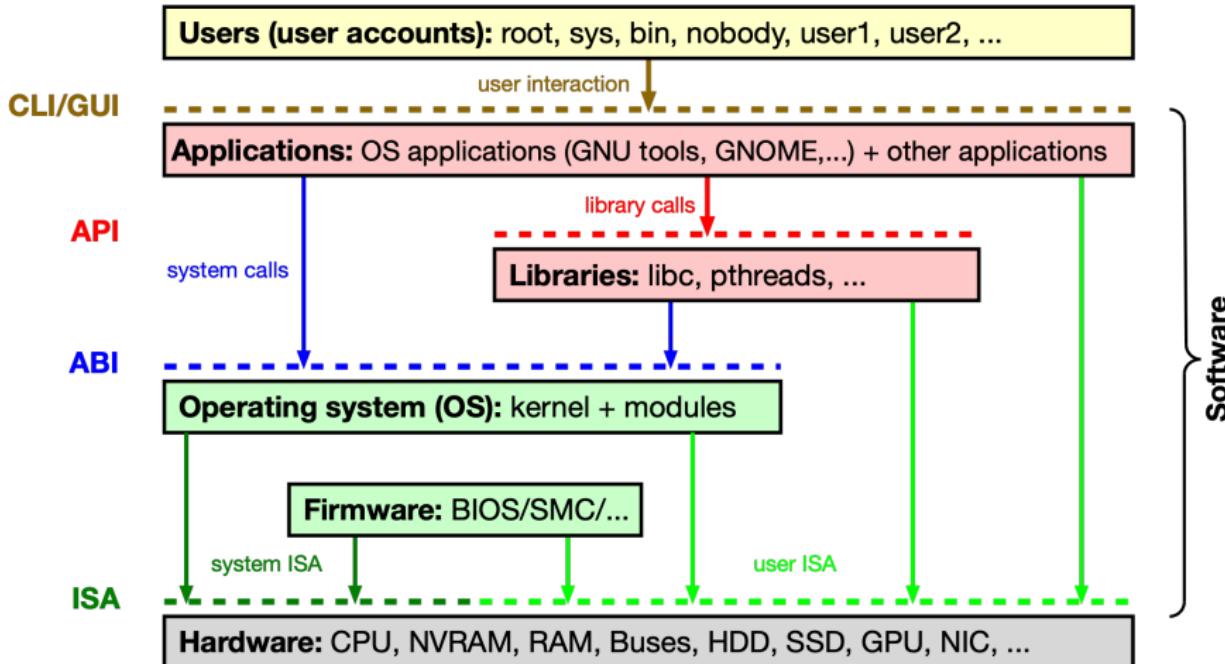
České vysoké učení technické v Praze,
Fakulta informačních technologií
Katedra počítačových systémů

<https://courses.fit.cvut.cz/BI-OSY>

Obsah přednášky

- 1 Model výpočetního systému
- 2 Uživatelé
- 3 Hardware
 - Procesor, paměť, sběrnice, periferní zařízení
- 4 Operační systém
 - Definice
 - Jádro, systémová volání
 - Vlastnosti moderních OS
- 5 Klasifikace výpočetních systémů
 - Jednojádrový systém
 - Vícejádrový systém se sdílenou pamětí (UMA, NUMA)
 - Vícejádrový systém s distribuovanou pamětí (cluster)

Model výpočetního systému



 Kernel mode

ISA = Instruction Set Architecture
(system ISA + user ISA)

API = Application Program Interface

 User mode

ABI = Application Binary Interface

CLI = Command Line Interface

GUI = Graphic User Interface

Uživatelé

- Uživatele můžeme rozdělit do několika kategorií, podle způsobu používání výpočetního systému.

► Uživatel

- ★ **běžný**: minimální znalosti OS, většinou přistupuje k systému přes GUI.
- ★ **pokročilý**: hlubší znalosti OS, pro práci se systémem využívá jak GUI, tak i CLI, umí vytvářet např. skripty (dávky) a využívá např. API pro práci s některými aplikacemi.

► Správce

- ★ **systémový**: instaluje, konfiguruje, spravuje HW a OS
- ★ **aplikiční**: instaluje, konfiguruje a spravuje jednotlivé aplikace

► Vývojář

- ★ **systémový**: vyvíjí samotné jádro nebo jednotlivé drivery OS
- ★ **aplikiční**: vyvíjí jednotlivé aplikace

Hardware: Procesor

- Central processing unit (CPU)
- Implementuje konkrétní Architekturu souboru instrukcí (Instruction set architecture = ISA), která definuje
 - ▶ množinu instrukcí, adresní režimy, ...
 - ▶ množinu registrů, organizaci paměti, organizaci V/V, ...

Příklad: Systémy s různými CPU a ISA.

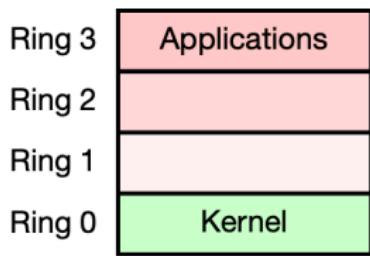
Systém	CPU	Výrobce	ISA	OS
A	AMD Ryzen 3 2200G	AMD	x86-64	Linux
B	Intel Core i5-8400	Intel	x86-64	Windows 10
C	Intel Xeon E3-1225 v6	Intel	x86-64	Linux
D	Ultra SPARC T2	Oracle	SPARC v9	Solaris

- ▶ CPU od různých výrobců (např. Intel a AMD) mohou implementovat stejnou ISA.
 - ★ Binární program ze systému A bude fungovat i na systému C (stejná ISA i OS).
 - ★ Binární program ze systému B nebude fungovat na ostatních systémech (různá ISA nebo OS) ⇒ možné zkompilovat, pokud kód nezávisí na ISA/OS.

Hardware: Procesor

- ISA také definuje různé privilegované úrovně (módy) běhu CPU
 - SW běžící v daném módu CPU může
 - používat pouze instrukce povolené v tomto módu,
 - modifikovat určité registry,
 - modifikovat určité oblasti paměti, ...
 - Pro běh OS jsou důležité dva základní módy
 - Kernel mód:** vše je povoleno, typicky v něm běží jádro OS.
 - User mód:** omezený mód (nelze přímo manipulovat s periferními zařízeními, ...), typicky v něm běží uživatelské procesy.

Příklad: Privilegované módy v procesorech od Intelu



- **Ring 0:** vše je povoleno.
- **Ring 3:** nelze
 - měnit aktuální ring,
 - měnit tabulku stránek,
 - registrovat interrupt handlery,
 - provádět V/V instrukce,...

- **Rozdělení procesorů podle typu instrukcí**

- ▶ CISC (Complex Instruction Set Computer)

- ★ Relativně velký počet komplexnější instrukcí.
 - ★ Obsahuje "memory-to-memory instrukce".
 - ★ Instrukce jsou různě dlouhé a doba jejich zpracování se liší
⇒ horší proudové zpracování instrukcí.
 - ★ Větší nároky na hardware (architekturu CPU).
 - ★ Program je obvykle reprezentován menším počtem instrukcí.
 - ★ Příklad ISA: x86-64.

- ▶ RISC (Reduced Instruction Set Computer)

- ★ Relativně malý počet jednoduchých instrukcí.
 - ★ Obsahuje hlavně "register-to-register" instrukce.
 - ★ Instrukce jsou stejně dlouhé a s podobnou dobou zpracování
⇒ lepší proudové zpracování instrukcí.
 - ★ Větší nároky na SW (překladač).
 - ★ Program je obvykle reprezentován větším počtem instrukcí.
 - ★ Příklad ISA: SPARC v9, ARM

Hardware: procesor

Příklad: Ukázka kódu pro násobení dvou čísel v paměti.

- Hodnoty dvou čísel jsou umístěny v paměti na adresách A1 a A2.
- Procesor obsahuje registry R1 a R2.
- **CISC architektura**

```
multm R1, A1, A2  
store A1, R1
```

- ▶ Instrukce `multm` načte hodnoty z paměti do registrů a vynásobí obsahy registrů a výsledek uloží do registru.
- ▶ Instrukce `store` uloží obsah registru do paměti.

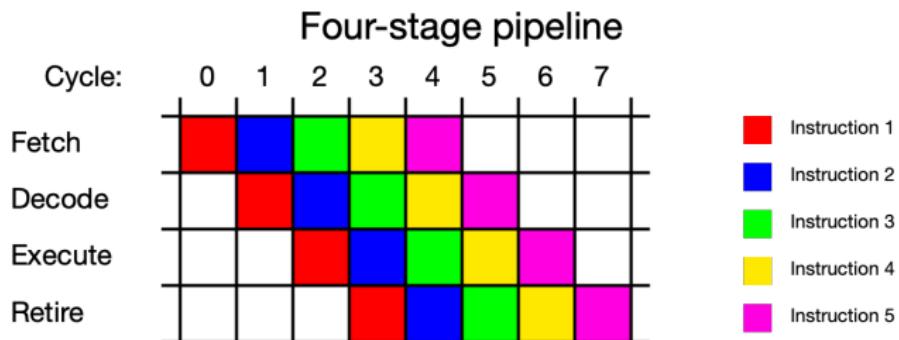
- **RISC architektura**

```
load  R1, A1  
load  R2, A2  
multr R1, R2  
store A1, R1
```

- ▶ Instrukce `load` načte hodnotu z paměti do registru.
- ▶ Instrukce `multr` vynásobí obsah registrů, výsledek uloží do registru.

• Zpracování instrukcí

- ▶ Probíhá obvykle v několika fázích v konkrétních částech CPU
 - 1 Fetch: načtení instrukce z paměti do CPU.
 - 2 Decode: dekódování instrukce a načtení operandů do registrů.
 - 3 Execute: zpracování instrukce.
 - 4 Retire: uložení výsledku.
- ▶ Aby byly všechny části CPU maximálně využívány se proudové zpracování instrukcí.



- ▶ Různé typy instrukcí trvají různě dlouho ⇒ CPU může používat několik proudových jednotek (pipelines) pro různě složité instrukce.

- **Více instrukčních proudových jednotek (instruction pipelines)**

- ▶ Load/Store pipe

- ★ Provádí instrukce pro načítání/ukládání dat z/do paměti.
 - ★ Doba zpracování instrukce závisí na aktuálním umístění dat (skryté paměti, hlavní paměť, systém souborů).

- ▶ Integer pipe

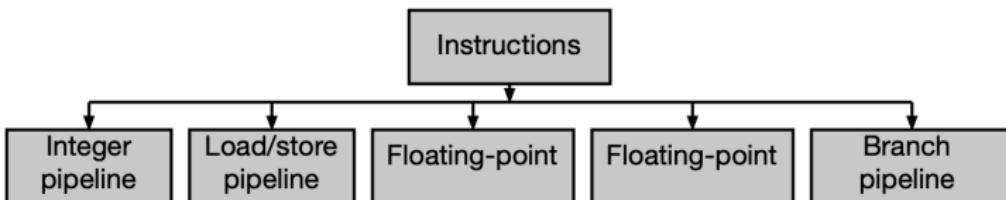
- ★ Zpracovává instrukce pro celočíselnou aritmetiku.

- ▶ Floating-point pipe

- ★ Složitější operace než s celými čísly ⇒ vyžadují více času.

- ▶ Branch pipe

- ★ Instrukce skoku způsobí načtení další instrukce z jiné oblasti paměti.
 - ★ Dva způsoby: "branching" (větvení výpočtu, např. If/else) nebo "calling" (volání a návrat ⇒ nutné si zapamatovat info pro návrat).



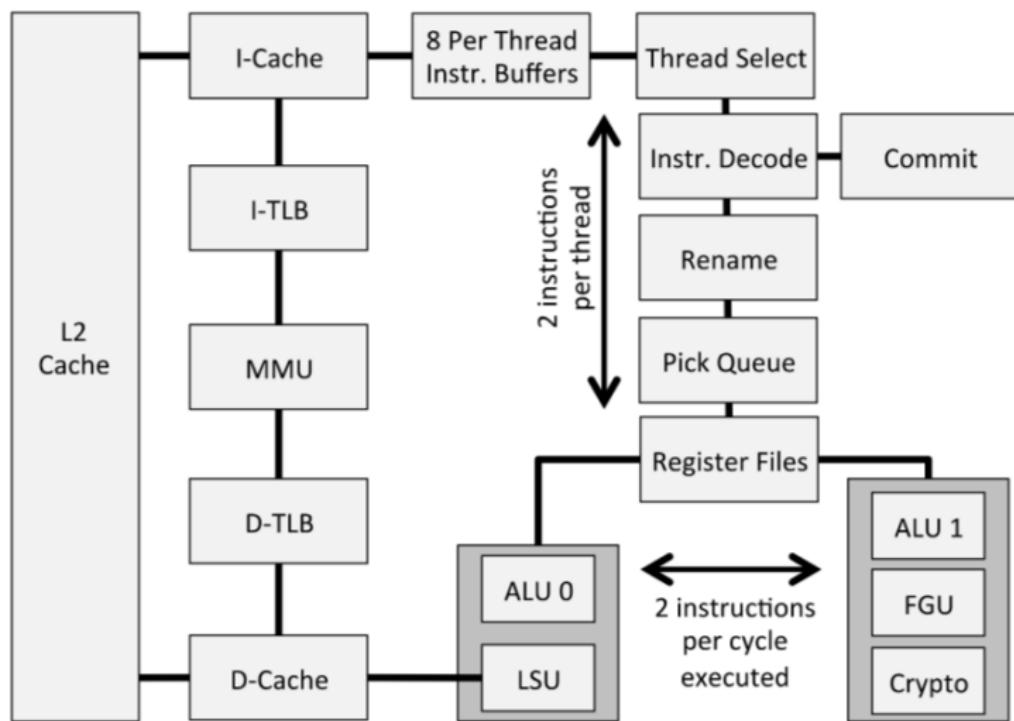
- **Rozdělení procesorů podle pořadí zpracování instrukcí**
 - ▶ **in-order-execution**
 - ★ Instrukce jsou zpracovávány v pořadí definovaném překladačem.
 - ★ Pokud instrukce používá výsledek z předchozí instrukce, který není ještě k dispozici, pak musí počkat na dokončení předchozí instrukce.
 - ★ "Správné" pořadí instrukcí je důležité ⇒ kvalitní překladač.
 - ★ Horší využití CPU.
 - ▶ **out-of-order-execution**
 - ★ Snaha minimalizovat čekání.
 - ★ Místo čekání na dokončení předchozí instrukce se CPU pokusí začít zpracovávat některou s dalších instrukcích, která není závislá na dokončení předchozích instrukcí ⇒ lepší využití, ale složitější architektura CPU.
 - ★ Příklad ISA: používáno v moderních CPU (x86-64, SPARC v9,...).

Hardware: procesor

- **Rozdělení procesorů podle počtu jader**
 - ▶ Jedno jádrový (single-core)
 - ★ CPU obsahuje pouze jedno jádro.
 - ★ Toto jádro zpracovává pouze jeden instrukční proud (jedno vláko).
 - ▶ Více jádrový (multi-core)
 - ★ CPU obsahuje více jader.
 - ★ Každé jádro umí zpracovávat minimálně jeden instrukční proud.
- **Některé CPU umožňují zpracovávat "současně" více instrukčních proudů jedním jádrem.**
 - ▶ Hyper-threading
 - ★ Technologie používaná v některých procesorech od Intelu.
 - ★ Kritické části jádra jsou zduplikované, tak aby jádro umožňovalo zpracovávat dva instrukční proudy (dvě vlákna).
 - ▶ Multi-threading
 - ★ Technologie používaná u procesorů řady Ultra SPARC T.
 - ★ Části každého jádra jsou zdvojené.
 - ★ Využívá proudového zpracování 2x4 instrukčních proudů (8 vláken).
 - ▶ Levnější varianta, jak zpracovávat více instrukčních proudů, aniž bychom museli přidat další jádra.
 - ▶ Efektivita je závislá na zpracovávaných instrukčních proudech.

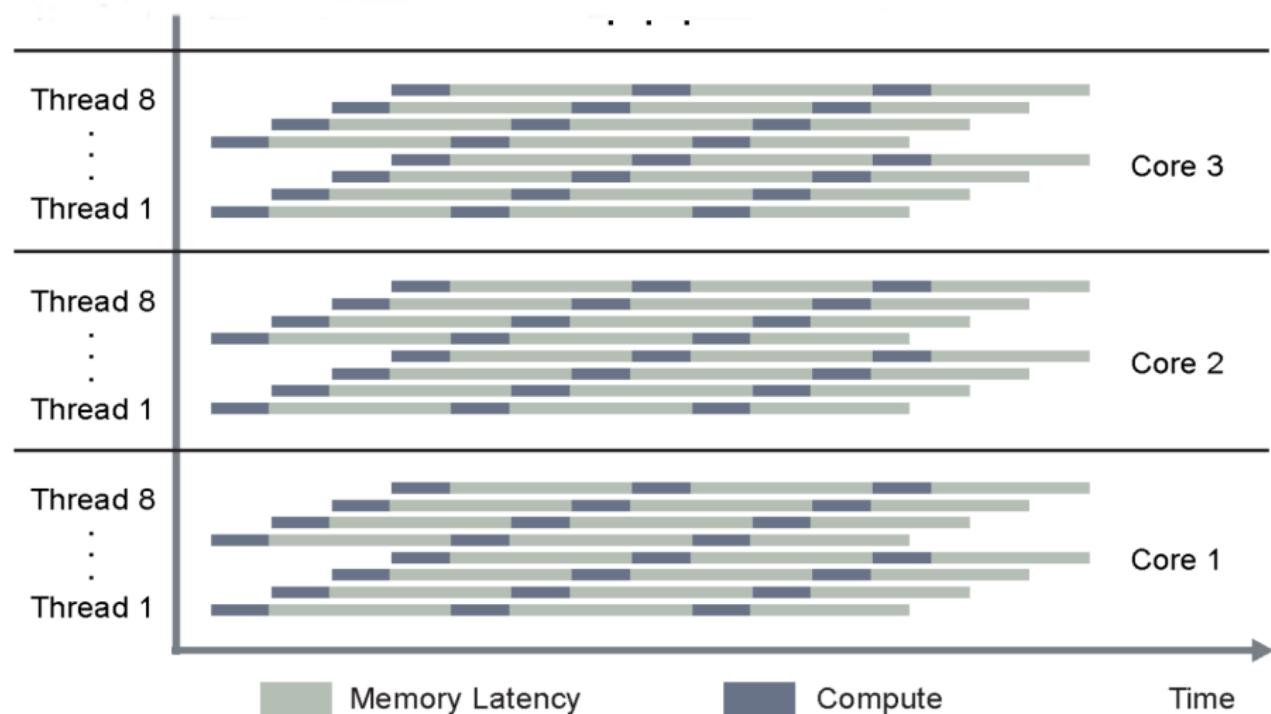
Hardware: procesor

Příklad: Blokový diagram SPARC S3 core



Hardware: procesor

Příklad: Procesor Ultra SPARC T4 – multi-threading



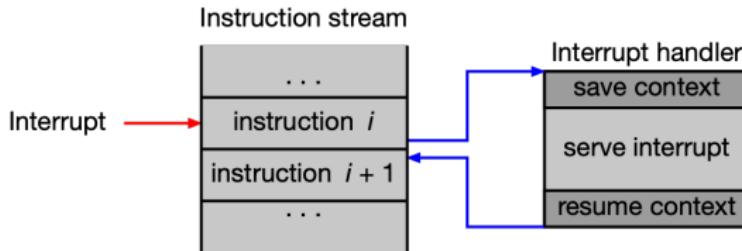
Hardware: procesor

• Přerušení (interrupts)

- ▶ Asynchronní reakce na nějakou událost.
- ▶ Normální zpracování instrukčního proudu jádra je přerušeno a začne se provádět obslužná rutina přerušení.
- ▶ Mechanismus přerušení je definován ISA.

• Kdy dochází k přerušení

- ▶ při zpracování instrukce
 - ★ přepnutí do jiného modu CPU, ...
 - ★ při chybě (např. dělení nulou, přetečení,...).
- ▶ když V/V zařízení žádá o pozornost
 - ★ generováno řadičem zařízení při dokončení operace,
 - ★ při chybě.



Paměť

- NVRAM (non-volatile random-access memory)/flash paměť
 - ▶ Paměť s přímým přístupem, drží informaci i při odpojeném napájení.
 - ▶ Obsahuje
 - ★ **proměnné**: např. specifikace zařízení, ze kterého se má spustit OS, ...
 - ★ **firmware**: detekce a konfigurace HW, spuštění OS (BIOS, Open Firmware, ...).
- RAM (random-access memory)
 - ▶ Paměť s přímým přístupem, drží informaci pouze při napájení.
 - ▶ Reprezentuje hlavní paměť, ve které se nachází jádro OS a spuštěné aplikace.

Sběrnice (bus)

- Zajišťuje přenos dat a řídicích povelů mezi jednotlivými částmi výpočetního systému (PCI-Express, Fibre Channel, SCSI, ...).

Periferní zařízení

Periferní zařízení

- Datová úložiště
 - ▶ Sekundární paměť pro záznam a čtení adresovatelných dat, drží informaci i při odpojeném napájení.
 - ▶ Reprezentována různými typy zařízení (HDD, SSD, RAID, ...).
- Síťové karty
- Grafické karty
- Monitory, klávesnice, myš, ...
- Zdroje, chlazení, ...

Operační systém (OS)

• Definice

- ▶ Základní SW, který funguje jako prostředník mezi HW a aplikacemi/uživateli.

• Úkoly

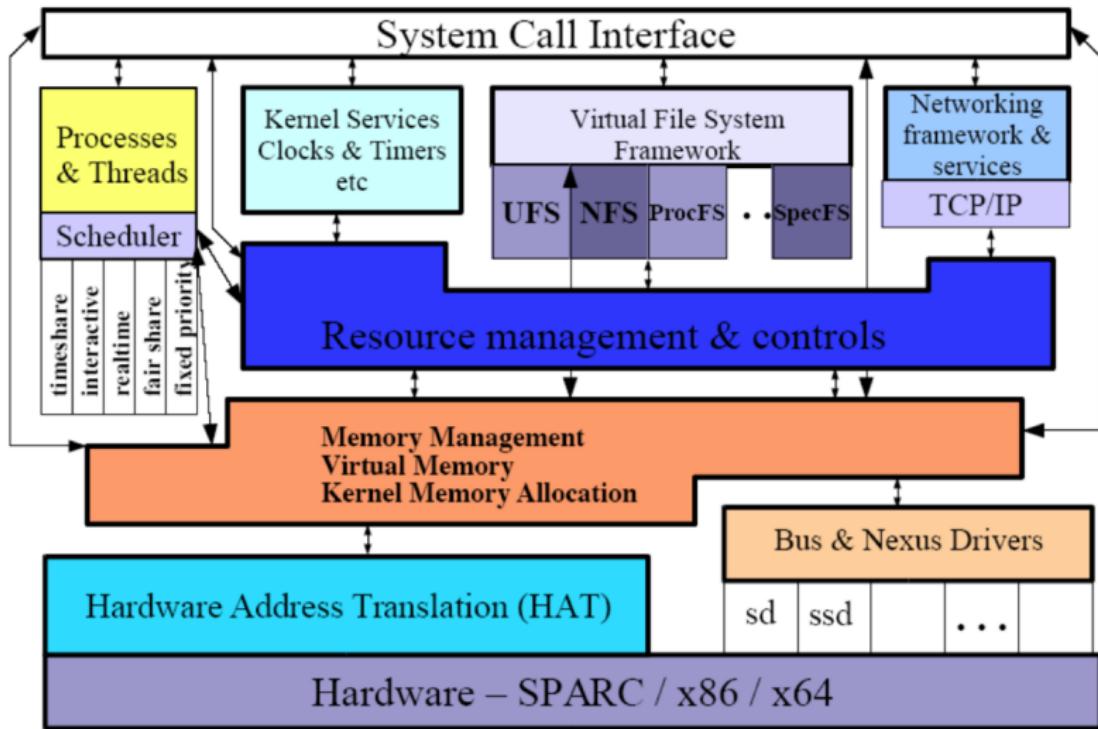
- ▶ Správa a sdílení výpočetních prostředků
 - ★ fyzických (procesor, paměť, disky, ...)
 - ★ logických (uživatelská konta, procesy, soubory, přístup. práva,...)
- ▶ Poskytuje rozhraní (abstrakce složitosti HW)
 - ★ aplikacím (Win32 API, Win64 API, systémová volání Unixu,...)
 - ★ uživatelům (CLI a GUI)

• V tomto předmětu se zaměříme na principy univerzálních OS

- ▶ MS Windows, OS unixového typu (Linux, Solaris, MacOS, BSD,...), VMS, ...

- OS jako celek je reprezentován
 - ▶ samotným jádrem OS (např. Linux kernel), které implementuje
 - 1 správu procesů a vláken (vytváření, plánování, ...),
 - 2 správu IPC prostředků (synchronizace, mezi procesová komunikace,...),
 - 3 správu paměti (alokace/dealokace, informace pro překlad adres,...),
 - 4 správu datových úložišť (RAID, HDD, SSD, USB flash, ...),
 - 5 správu systému souborů,
 - 6 správu V/V (přerušení, ovladače, ...)
 - 7 správu síťové infrastruktury (síťová rozhraní, IP stack, ...),...
 - ▶ aplikacemi, které jsou součástí OS a které implementují
 - ★ CLI (např. GNU nástroje, PowerShell,...),
 - ★ GUI (např. X11, GNOME, KDE, ...),
 - ★ správu SW (instalce aplikace,...),
 - ★ správu uživatelů, ...
- V tomto předmětu se zaměříme problematiku související s body 1-5.

- **Příklad:** architektura jádra Solarisu



Systémová volání (syscalls)

• Aplikace

- ▶ typicky běží v user modu CPU
 - ⇒ nemohou přímo používat prostředky systému,
- ▶ pokud chce použít prostředek systému
 - ⇒ musí požádat jádro OS pomocí systémového volání.
 - ★ Konkrétní mechanismus systémového volání je závislý na ISA procesoru a použitém OS.

• Příklad: možný mechanismus systémového volání

- ▶ Systémové volání `write(int fd, void *buf, int N)`
 - ★ zapíše N bytů z paměti `buf` na soubor určeným deskriptorem `fd`.
- ▶ Aplikace
 - ★ uloží parametry systémového volání `fd`, `buf` a `N` (např. na zásobník, registrů,...),
 - ★ uloží informaci o požadovaném systémovém volání,
 - ★ vyvolá softwarové přerušení pomocí příslušné instrukce, která způsobí přepnutí CPU do kernel modu a spustí obslužnou rutinu v jádře OS.
- ▶ Jádro
 - ★ na základě požadovaného systémového volání provede příslušné operace (ověří přístupová práva, pokusí se zapsat N bytů,...).

Systémová volání (syscalls)

- **Příklad:** Informace o systémových volání v různých OS
 - ▶ Linux: Manualové stránky: `man 2 intro` nebo
 - ▶ Solaris: Oracle Docs: System Calls
 - ▶ MS Windows: Microsoft Docs: Windows API

- **Příklad:** Sledování systémových volání volaných z aplikace
 - ▶ Linux

```
linux:~> strace date
execve("/usr/bin/date", ["date"], 0x7ffe540997c0 /* 88 vars */) = 0
...
write(1, "Fri Feb 15 09:14:46 CET 2019\n", 29) = 29
...
```
 - ▶ Solaris

```
frayl:~> truss date
execve("/usr/bin/date", 0xFE01BB2C, 0xFE01BB34)  argc = 1
...
write(1, " F r i   F e b   1 5   0"..., 29)      = 29
_exit(0)
```

- Licencování OS
 - ▶ Různé podmínky pro používání, šíření, modifikaci a různá dostupnost zdrojového kódu OS (Open software, Libre software,...).
- Víceúlobový (multitasking, time-sharing)
 - ▶ Běh více úloh (procesů) se sdílením času.
 - ▶ Ochrana paměti, plánování procesů/vláken.
- 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 jádrech CPU.
- Unifikované prostředí
 - ▶ Přenositelnost mezi platformami (90% jádra v jazyce C).

OS a různé typy výpočetních systémů

- Vestavěné zařízení (embedded device)
 - ▶ OS: Linux, proprietární OS, ...
- Chytré zařízení
 - ▶ OS: Adroind, iOS, watchOS, tvOS, ...
- Laptop, stolní počítač
 - ▶ OS: MS Windows, MacOS, Linux, ChromeOS, ...
- Server
 - ▶ OS: unixového OS (Linux, Solaris, HP-UX, AIX), MS Windows, ...
 - ▶ Příklad konfigurace: [Oracle](#)
- Super počítače
 - ▶ OS: unixového OS (Linux, ...), proprietární OS, ...
 - ▶ Příklady konfigurací: www.top500.org
- **Příklad:** Zastoupení OS na různých zařízeních

Mobile 2019-01 (netmarketshare.com)

Android	70,64 %
iOS	28,15 %
Others	1,03 %
Windows phone	0,06 %

Desktop 2019-01 (netmarketshare.com)

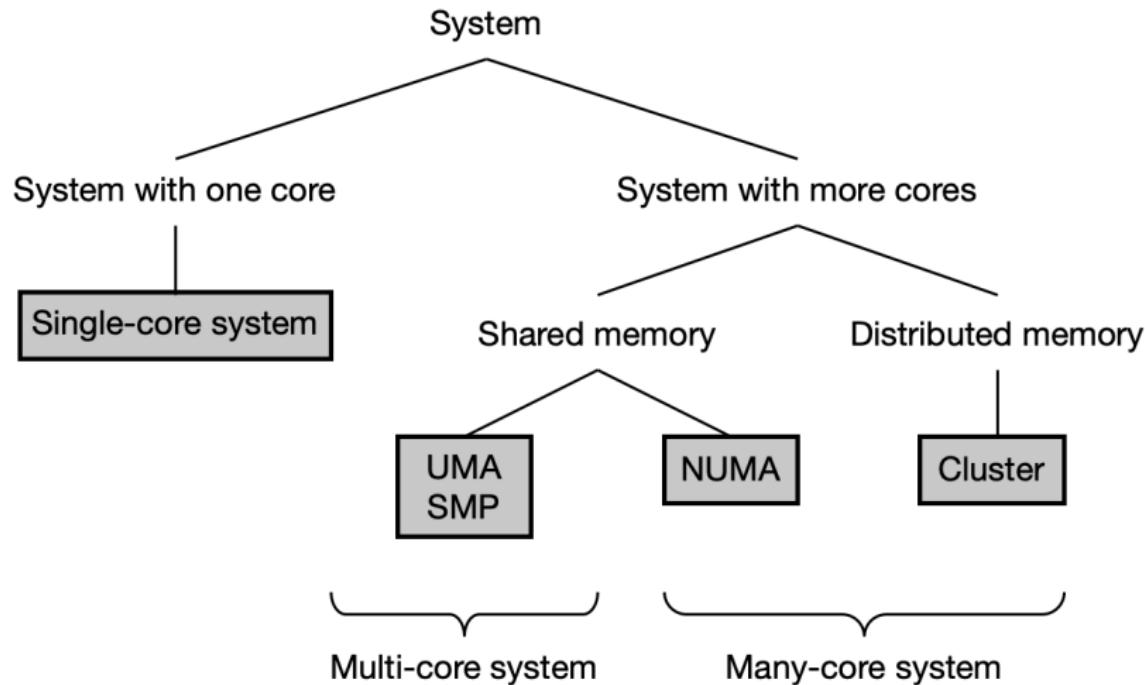
Windows	86,23 %
MacOS	10,59 %
Linux	2,4 %
Chrome OS	0,37 %
Others	0,41 %

Websites 2019-01 (w3techs.com)

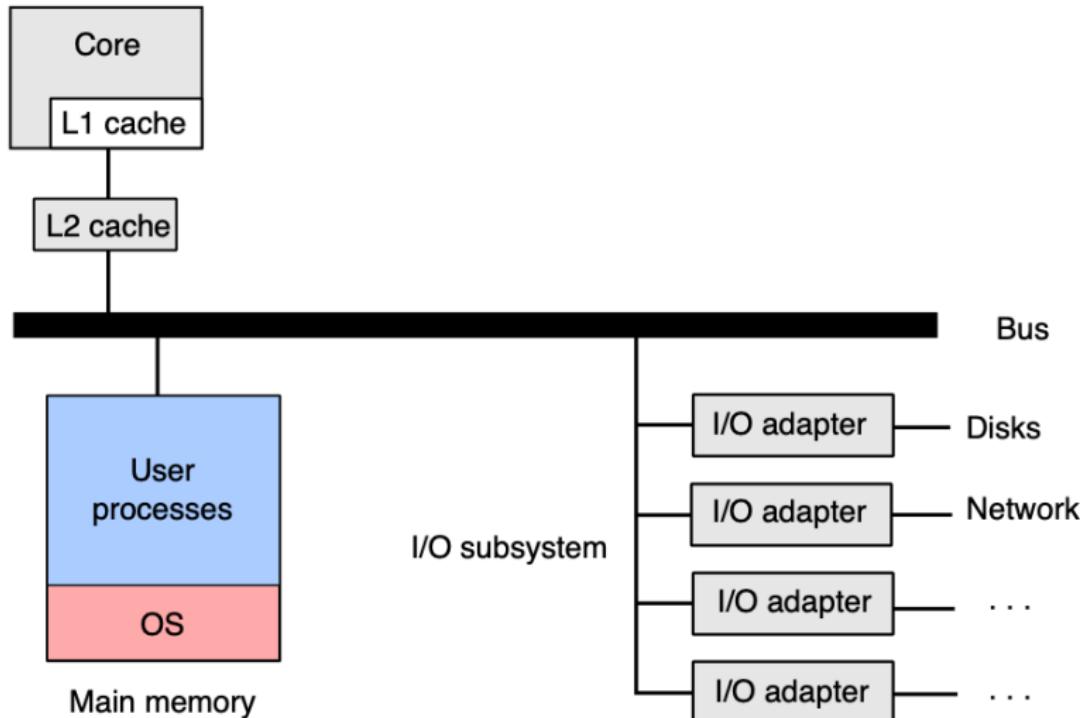
Linux	37,00 %
Windows	30,70 %
Others	32,3 %

Klasifikace výpočetních systémů

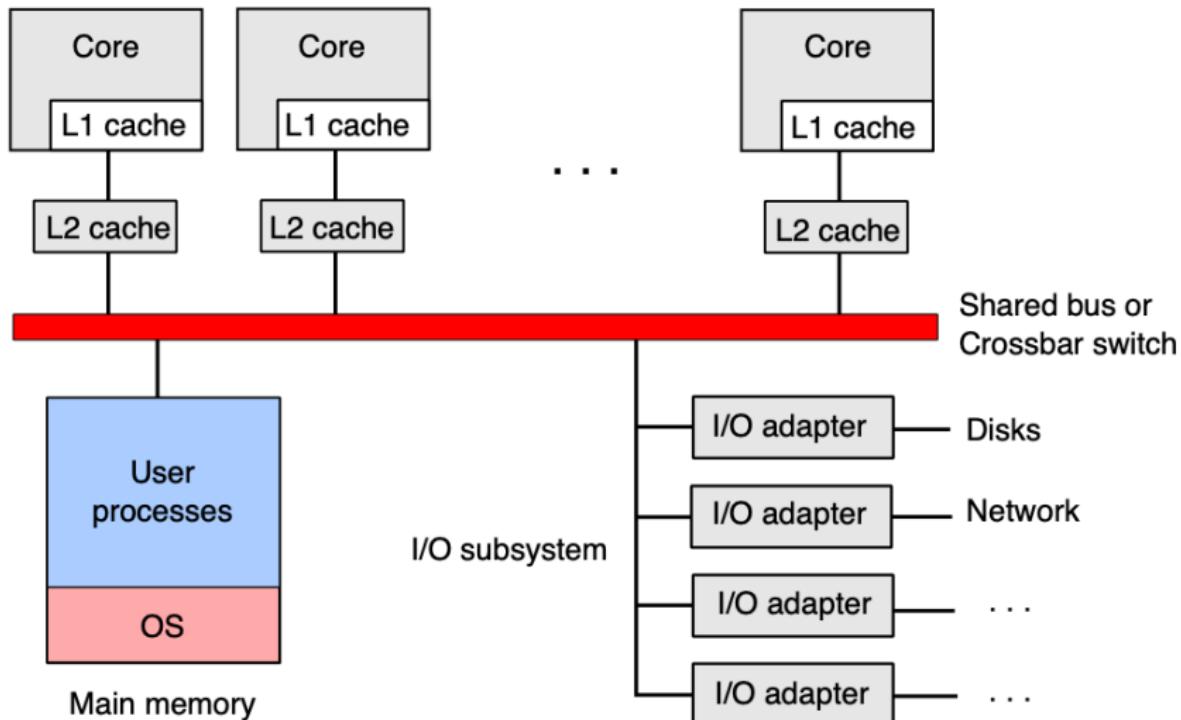
- Podle počtu jader a způsobu přístupu k paměti



Jedno jádrový systém



UMA - uniform memory access

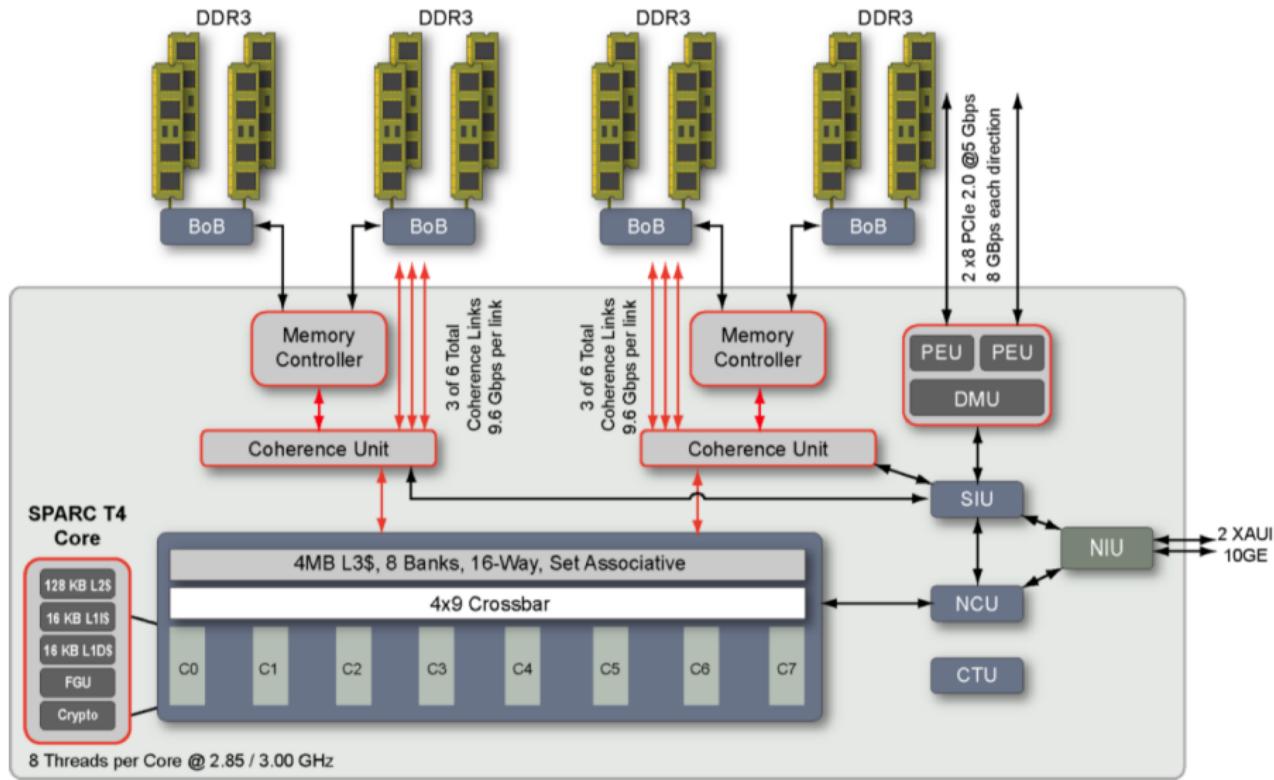


UMA - uniform memory access

- Systém se skládá z několika identických jader.
- Jádra jsou propojena sdílenou sběrnicí nebo propojovací sítí.
- Jádra sdílí stejnou paměť (jeden nebo několik modulů).
- Přístupový čas (*latence*) do paměti je pro všechny jádra téměř stejný.
- Všechny jádra sdílí I/O.
- Tato HW architektura je také označována jako **SMP (symmetric multiprocessors)**.
- Sdílená sběrnice je slabé místo, proto může být nahrazena propojovací sítí (crossbar switch).

UMA - uniform memory access

Příklad: Procesor Ultra SPARC T4



UMA - uniform memory access

Příklad:

- Informace o serveru `fray1.fit.cvut`

- ▶ ISA

```
fray1:~> uname -a
SunOS fray1 5.11 11.3 sun4v sparc SUNW,SPARC-Enterprise-T5120
```

- ▶ CPU

```
fray1:~> psrinfo -pv
The physical processor has 4 cores and 32 virtual processors (0-23,32-39)
  The core has 8 virtual processors (0-7)
  The core has 8 virtual processors (8-15)
  The core has 8 virtual processors (16-23)
  The core has 8 virtual processors (32-39)
    UltraSPARC-T2 (chipid 0, clock 1165 MHz)
```

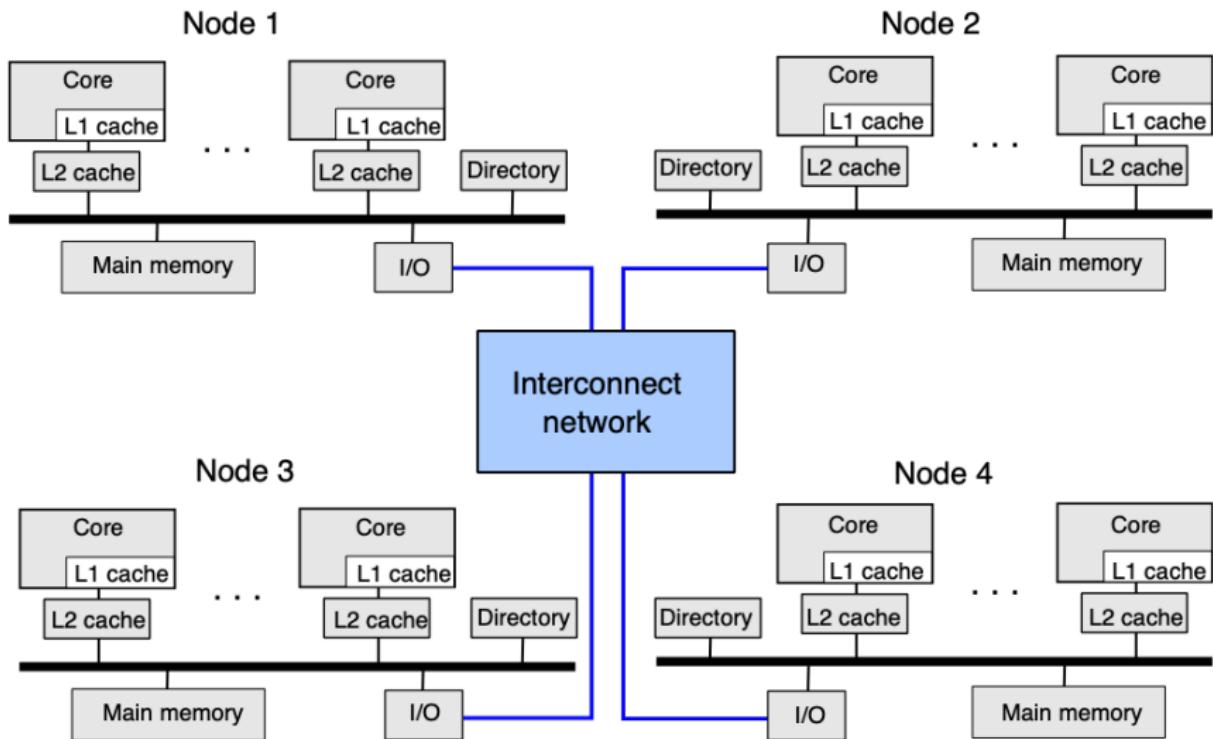
- ▶ OS

```
fray1:~> cat /etc/release
                                         Oracle Solaris 11.3 SPARC
Copyright (c) 1983, 2015, Oracle and/or its affiliates. All rights reserved.
Assembled 06 October 2015
```

- Informace o serveru s OS Linux

- ▶ Pomocí příkazů: `uname -a`, `lscpu`, `cat /etc/os-release`.

NUMA - nonuniform memory access

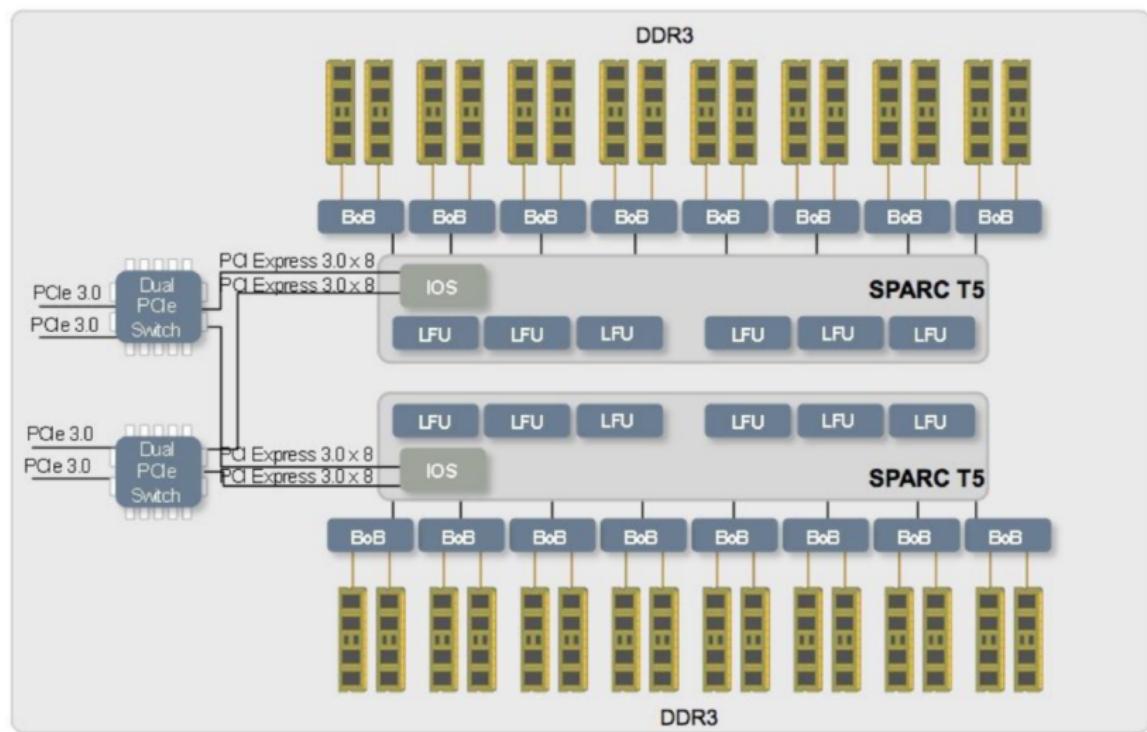


NUMA - nonuniform memory access

- Lokální paměť uzlu je viditelná pro jádra z ostatních uzlů.
- Přístup do vzdálené paměti je pomalejší než do lokální paměti.
- Koherence skrytých pamětí je zajištěna hardwarem, proto je tato architektura někdy označována jako cache-coherent NUMA.
- OS má znalost o paměťových latencích mezi jednotlivými jádry a pamětími, a může ji využívat při plánování vláken.

NUMA - nonuniform memory access

- Příklad: Dual-socket SPARC T5 konfigurace



NUMA - nonuniform memory access

Příklad:

- Informace o serveru `fray2.fit.cvut`

- ▶ CPU

```
fray2:~> psrinfo -vp
The physical processor has 8 cores and 64 virtual processors (0-63)
    The core has 8 virtual processors (0-7)
    The core has 8 virtual processors (8-15)
    The core has 8 virtual processors (16-23)
    The core has 8 virtual processors (24-31)
    The core has 8 virtual processors (32-39)
    The core has 8 virtual processors (40-47)
    The core has 8 virtual processors (48-55)
    The core has 8 virtual processors (56-63)
        UltraSPARC-T2+ (chipid 0, clock 1414 MHz)
The physical processor has 8 cores and 64 virtual processors (64-127)
    The core has 8 virtual processors (64-71)
    The core has 8 virtual processors (72-79)
    The core has 8 virtual processors (80-87)
    The core has 8 virtual processors (88-95)
    The core has 8 virtual processors (96-103)
    The core has 8 virtual processors (104-111)
    The core has 8 virtual processors (112-119)
    The core has 8 virtual processors (120-127)
        UltraSPARC-T2+ (chipid 1, clock 1414 MHz)
```

NUMA - nonuniform memory access

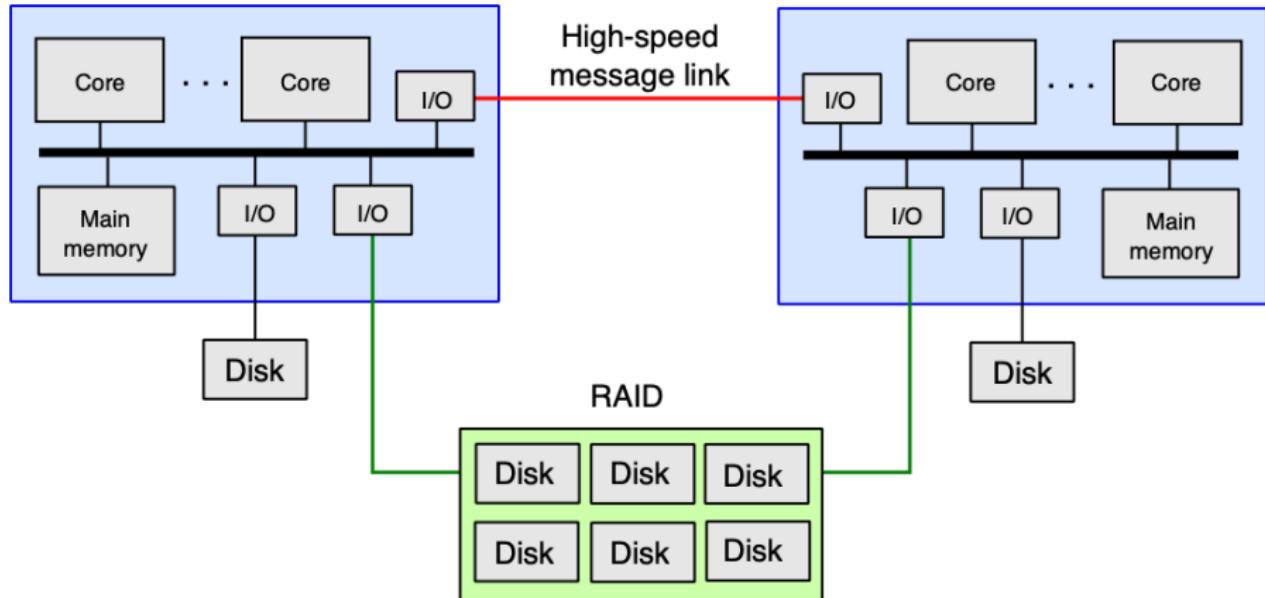
- ▶ Locality group (lgroup): info o paměťových latencích mezi uzly

```
fray2:~> lgrpinfo
lgroup 0 (root):
    Children: 1 2
    CPUs: 0-127
    Memory: installed 32G, allocated 11G, free 20G
    Lgroup resources: 1 2 (CPU); 1 2 (memory)
    Latency: 18
lgroup 1 (leaf):
    Children: none, Parent: 0
    CPUs: 0-63
    Memory: installed 16G, allocated 4.3G, free 11G
    Lgroup resources: 1 (CPU); 1 (memory)
    Load: 0.0188
    Latency: 12
lgroup 2 (leaf):
    Children: none, Parent: 0
    CPUs: 64-127
    Memory: installed 16G, allocated 7.2G, free 8.8G
    Lgroup resources: 2 (CPU); 2 (memory)
    Load: 0.0118
    Latency: 12
```

- ▶ Informace o procesech jejich přiřazení do lgroup

```
fray2:~> ps -H
 PID LGRP TTY      TIME CMD
 25351    2 pts/27    0:00 ps
 25008    2 pts/27    0:00 bash
```

Cluster



Cluster

- Základní uzel
 - ▶ CPU, lokální paměť, síťové rozhraní, disk,...
 - ▶ Ostatní periferie mohou chybět.
 - ▶ OS + SW podporující některé funkce clusteru (vyvažování výkonu,...).
- Uzly jsou propojeny vysokorychlostní propojovací sítí.
- Použití
 - ▶ Výpočetní systémy.
 - ▶ Fault-tolerant systémy.
 - ▶ Systémy s vyvažováním zátěže.
- Typy podle konfigurace
 - ▶ Homogenní
 - ★ Uzlu mají stejnou/podobnou hardwarovou konfiguraci.
 - ★ Na všech uzlech je stejný OS.
 - ▶ Heterogenní
 - ★ Uzly mají různé hardwarové konfigurace.
 - ★ Na jednotlivých uzlech se používají různé OS.

- ① A. S. Tanenbaum, H. Bos: *Modern Operating Systems (4th edition)*, Pearson, 2014.
- ② W. Stallings: *Operating Systems: Internals and Design Principles (9th edition)*, Pearson, 2017.
- ③ A. Silberschatz, P. B. Galvin, G. Gagne: *Operating System Concepts (9th edition)*, Wiley, 2012.
- ④ R. Buyya: *Mastering Cloud Computing: Foundations and Applications Programming (1st Edition)*, Morgan Kaufmann, 2013.
- ⑤ D. Gove: *Solaris Application Programming (1st Edition)*, Prentice Hall, 2008.
- ⑥ Oracle's SPARC T4-1, SPARC T4-2, SPARC T4-4, and SPARC T4-1B Server Architecture, Oracle White Paper, 2012.
- ⑦ *READ_ME_FIRST: What Do I Do with All of Those SPARC Threads?*, Oracle Technical White Paper, 2013.

Operační systémy

Procesy a vlákna. Časově závislé chyby. Kritické sekce.

Jan Trdlička



České vysoké učení technické v Praze, Fakulta informačních technologií
Katedra počítačových systémů

<https://courses.fit.cvut.cz/BI-OSY>

Obsah přednášky

1 Program

2 Proces

- Definice, vytvoření a ukončení

3 Vlákno

- Definice, vytvoření a ukončení

4 Multitasking/Multithreading

5 Plánování vláken, přepínání kontextu a stavy vláken

6 Časově závislé chyby a kritické sekce

7 Korektní paralelní program

Program(aplikace)

- Program je v systému reprezentován spustitelným binárním programem, který je uložený v sekundární paměti (např. disk).
- **Spustitelný binární program**
 - ▶ Formát je závislý na OS, pro který je zkompilovaný.
 - ★ Executable and Linkable Format (**ELF**) pro OS unixového typu, ...
 - ★ Portable Executable format (**PE/PE32+**) pro MS Windows, ...
 - ▶ Obsahuje
 - ★ **TEXT** = spustitelný binární kód,
 - ★ **DATA** = proměnné a jejich hodnoty, ...
 - ★ další informace (např.o **sdílených knihovnách**, ...).

Příklad: Určení formátu a informace o knihovnách

```
linux:~> file /usr/bin/date
/usr/bin/date: ELF 64-bit LSB shared object, x86-64, ...
dynamically linked, ..., stripped
```

```
linux:~> ldd /usr/bin/date
linux-vdso.so.1 (0x00007ffd66fc3000)
 libc.so.6 => /lib64/libc.so.6 (0x00007f6e3c5b1000)
 /lib64/ld-linux-x86-64.so.2 (0x00007f6e3cb84000)
```

Proces

- Instance spuštěného programu/aplikace.
- Entita, v rámci které jsou alokovány prostředky (paměť, vlákna, otevřené soubory, zámky, semafory, sokety,...).
- Implicitně každý proces má jedno výpočetní "main" vlákno.
- Jádro OS si pro každý proces udržuje celou řadu datových struktur nezbytných pro
 - ▶ identifikaci: číslo procesu(PID), číslo rodič. procesu(PPID),...
 - ▶ bezpečnost: identita procesu (USER, RUSER),...
 - ▶ správu paměti: informace pro překlad virt. adres(page table),...
 - ▶ správu FS: tabulka deskriptorů souborů,...
- Při vzniku nového procesu, část datových struktur je zděděna od rodičovského procesu (tabulka deskriptorů souborů,...) a část je nastavena na nové hodnoty, které jsou specifické pro nový proces (číslo procesu,...).

Proces s jedním vláknem

```
#include <stdio.h>

int sum(int a, int b)
{
    int s;
    s = a + b;
    return (s);
}

int main ( int argc, char * argv[ ] )
{
    int a, b;
    sscanf ( argv[1], "%d", &a);
    sscanf ( argv[2], "%d", &b);

    printf("Sum = %d\n", sum(a,b));

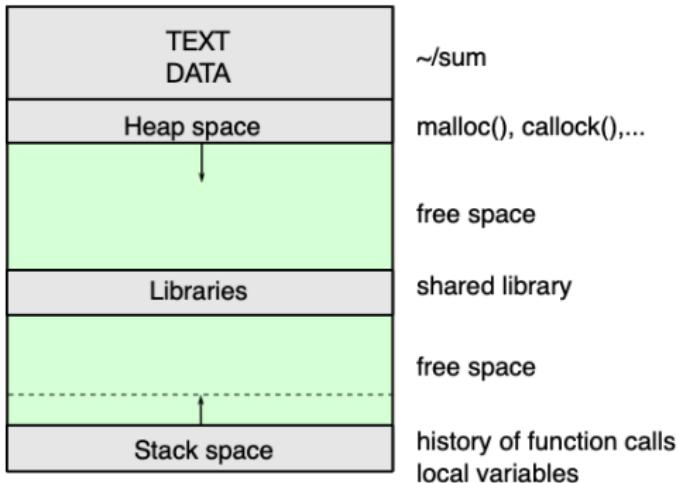
    return 0;
}
```

Program: sum.c

Kernel structures

PID,PPID, USER, RUSER, ...
page table, open files, ...
TID1,

Process address space



Process: ~/sum

Proces s jedním vláknem

Příklad: Zobrazení adresového prostoru procesu s číslem 5652 v OS unixového typu.

Address	Kbytes	RSS	Anon	Locked	Mode	Mapped File
0000000010000000	1000	1000	-	-	r-----	bash
000000001001FA000	48	48	16	-	rwx----	bash
00000000100300000	16	16	16	-	rw----	bash
000000CC71630000	256	256	192	-	rw----	[heap]
00007CBD99000000	232	208	-	-	r-----	libcurses.so.1
00007CBD9913A000	32	32	-	-	rwx----	libcurses.so.1
00007CBD99142000	16	16	-	-	rwx----	libcurses.so.1
00007CBD99200000	32	24	-	-	r-----	libgen.so.1
00007CBD99308000	8	8	-	-	rwx----	libgen.so.1
00007CBD99400000	64	64	-	-	rwx----	[anon]
00007CBD99500000	64	64	-	-	rwx----	[anon]
00007CBD9952A000	8	8	-	-	rwxs---	[anon]
00007CBD99600000	24	16	8	-	rwx----	[anon]
00007CBD99700000	1560	1288	-	-	r-----	libc.so.1
00007CBD99986000	64	64	16	-	rwx----	libc.so.1
00007CBD99996000	8	8	-	-	rwx----	libc.so.1
00007CBD99A00000	64	64	64	-	rw----	[anon]
00007CBD99A30000	8	8	-	-	r-s---	[anon]
00007CBD99B00000	272	272	-	-	r-----	ld.so.1
00007CBD99C44000	24	24	16	-	rwx----	ld.so.1
FFFFFD6DA3C10000	56	56	8	-	rw----	[stack]

total Kb	3856	3544	336	-		

- Výpočetní entita (proud instrukcí), které je přidělováno jádro CPU.
- Historicky se vlákno nazývalo "light-weight process" (lwp).
- Vlákna vytvořená v rámci procesu sdílí většinu prostředků alokovaných v tomto procesu.
- Jádro OS si pro každé vlákno udržuje celou řadu datových struktur nezbytných pro
 - ▶ identifikaci: číslo vlákna(TID),...
 - ▶ zásobník: lokální proměnné, historie volání,...
 - ▶ informace nutné pro přepínání kontextu: čítač instrukcí, aktuální hodnoty registrů,...
 - ▶ informace pro plánování vláken: priorita, čas strávený na CPU,...
 - ...

Proces s více vlákny

```
#include <pthread.h>
...
void *code(void * dummy)
{
    printf("Thread: Hello.\n");
    sleep(60);
    return(NULL);
}

int main ( int argc, char * argv[] )
{
    void *dummy;
    pthread_t tid1;
    pthread_t tid2;

    pthread_create(&tid1, NULL, &code, NULL);
    pthread_create(&tid2, NULL, &code, NULL);

    pthread_join(tid1, &dummy);
    pthread_join(tid2, &dummy);

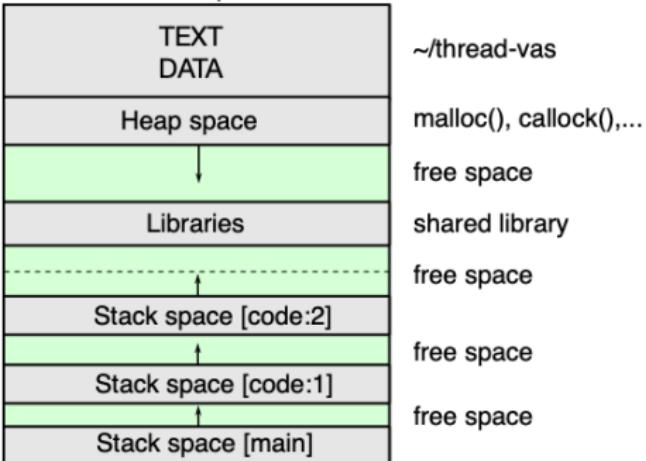
    return 0;
}
```

Program: thread-vas.c

Kernel structures

PID,PPID, USER, RUSER,...
page table open files, ...
TID1, TID2, TID3, ...

Process address space



Process: ~/thread-vas

Vytvoření nového procesu

- Nový proces se vytvoří, když existující proces zavolá příslušné systémové volání
 - ▶ v OS unixového typu: `fork(2)`, `execve(2)`, ...
 - ▶ v MS Windows: `CreateProcessA()`, ...
- Nově vzniklý proces může představovat
 - ▶ **kopii/klon původního procesu** (např. po zavolání `fork(2)`)
 - ★ Jádro alokuje nové dat. struktury pro nový proces (část z nich bude nově zinicializována, část bude kopií od rodiče).
 - ★ Adresový prostor procesu bude zkopirován od rodiče (TEXT, DATA, zásobník, halda, ...).
 - ★ Čítač instrukcí bude ukazovat na následující instrukci za `fork(2)`.
 - ▶ **úplně nový proces** (např. po zavolání `fork(2)` a `execve(2)`)
 - ★ Jádro alokuje nové dat. struktury pro nový proces (část z nich bude nově zinicializována, část bude kopií od rodiče).
 - ★ Adresový prostor procesu bude nově zinicializován (prázdný zásobník, halda, ...) a TEXT, DATA, knihovny budou načteny ze souboru.
 - ★ Čítač instrukcí bude ukazovat na první instrukci programu.

Příklad: Vytvoření nového procesu v Unixu

```
1 ...  
2 int main()  
3 {  
4     pid_t pid;  
5     int status;  
6  
7     pid = fork();  
8  
9     switch (pid) {  
10  
11         case -1: /* Error */  
12             exit(1);  
13  
14         case 0: /* Child process */  
15             char* argv[] = { "sleep", "30", NULL };      /* array of argument strings */  
16             char* envp[] = { NULL };                  /* array of environment strings */  
17             execve("/bin/sleep", argv, envp);  
18             exit(0);  
19  
20         default: /* Parent process */  
21             wait(&status);  
22     }  
23  
24     return 0;  
25 }
```

Příklad: Vytvoření nového procesu v Unixu

`fork()`

- Vytvoří nový proces, který je kopíí procesu, z kterého byla tato funkce zavolána.
- Funkce vrací
 - ▶ v rodičovském procesu číslo potomka (v případě chyby -1),
 - ▶ v potomkovi vždy hodnotu 0.
- Po návratu z funkce, rodič i potomek pokračuje na následující instrukci a běží na sobě "nezávisle".

`execve(const char *filename, ...)`

- Adresový prostor procesu, ze kterého je funkce volána, je přepsán obsahem souboru, který se začne vykonávat od začátku.

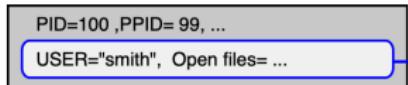
`wait(int *status)`

- Funkce zablokuje rodičovský proces, ve kterém je zavolána, dokud se jeden jeho potomek neukončí.
- Varianta `waitpid(2)` umožňuje čekat na konkrétního potomka.

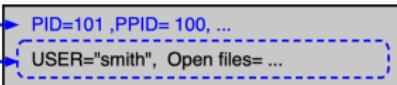
Více informací viz. `man 2 fork`, `man 2 execve`, `man 2 wait`.

Příklad: Vytvoření nového procesu v Unixu

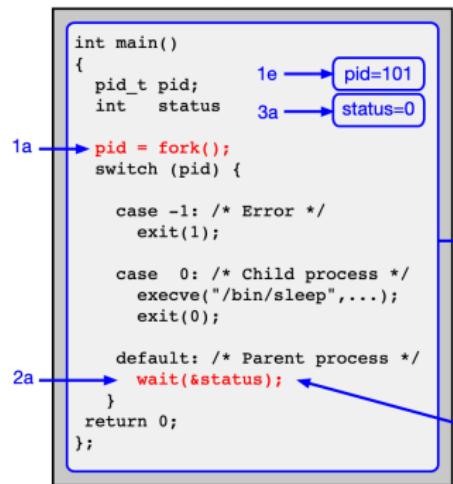
Kernel structures



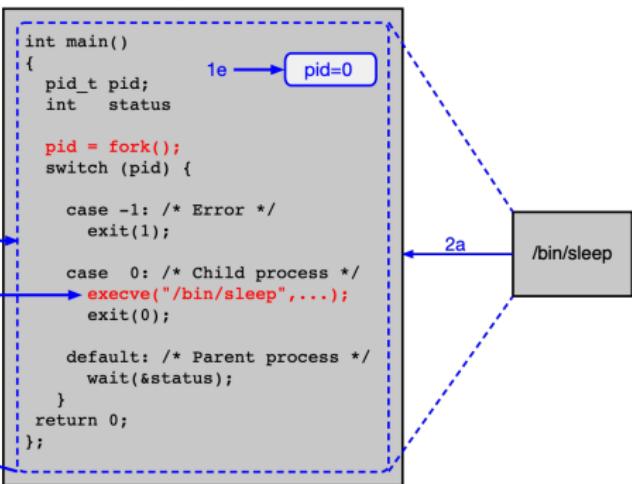
Kernel structures



Process address space



Process address space



Parent process

Child process

Příklad: Vytvoření nového procesu v Unixu

- Komentář k předchozímu obrázku

- 1a Volání `fork()` v rodiči způsobí alokování dat. struktur v jádře a paměti pro nový proces.
- 1b Do některých datových struktur nového procesu jsou nastaveny nové hodnoty.
- 1c Do dalších datových struktur nového procesu jsou zkopirovány hodnoty z rodiče.
- 1d Obsah adresového prostoru rodiče se "zkopíruje" do adresového prostoru potomka.
- 1e Při návratu z `fork()` se do proměnné `pid`
 - ★ v rodiči uloží číslo potomka a pokračuje se následující instrukci,
 - ★ v potomkovi uloží hodnota 0 a pokračuje se následující instrukci.
- 2a ★ Rodič bude čekat na dokončení potomka pomocí volání `wait()`.
★ Potomek načte nový program do svého adresového prostoru pomocí volání `execve()`.
- 3a Po ukončení potomka (např. pomocí funkce `exit()` nebo `return`) bude návratový kód (parametr těchto funkcí) uložen v rodiči do proměnné `status`.
- 3b Rodič se vráti z funkce `wait()` a pokračuje následující instrukcí.

Ukončení procesu

• Jádro OS při ukončení procesu

- ▶ se pokusí předat "návratový kod" rodiči,
- ▶ ukončí všechna vlákna, která existují v rámci daného procesu,
- ▶ uvolní adresový prostor procesu a příslušné dat. struktury v jádře související s daným procesem.

• Varianty ukončení procesu

- ▶ Proces se ukončí sám
 - ★ pokud dojde na konec programu (např. `return` v "main" vláknu),
 - ★ zavolá příslušnou knihovní funkci (např. `exit(3)`,
`TerminateProcess()`, ...).
- ▶ Proces je ukončen jádrem
 - ★ pokud dojde k fatální chybě (např. dělní nulou, špatná manipulace s pamětí,...),
 - ★ na základě např. přijatého signálu,...

Příklad: Vytváření nových procesů

- Kolik vznikne celkem procesů, pokud spustíme následující program?
- Kdy se ukončí poslední z nich?

```
1 ...
2 int main()
3 {
4     pid_t pid;
5     int    i;
6
7     for ( i = 0; i < 3; i++ )
8     {
9         pid = fork();
10
11         if ( pid == 0 )
12         {
13             sleep(10);
14         }
15     }
16
17     return 0;
18 }
```

Vytvoření a ukončení vlákna

- Procesy se implicitně vytváří s jedním "main" vlákнем.
- Pokud chceme vytvořit v rámci procesu další vlákna, pak můžeme použít
 - ▶ OS API/knihovny
 - ★ proprietární knihovny jednotlivých OS ([Microsoft Windows API](#), [Solaris thread library](#),...),
 - ★ [POSIX thread library](#) (MS Windows, GNU/Linux, FreeBSD, ...),
 - ★ [OpenMP](#) (MS Windows, GNU/Linux, FreeBSD, Solaris, ...),...
 - ▶ Programovací jazyky s vestavěnou podporou vláken
 - ★ [C++](#) (od C++11), Java,...

Příklad: POSIX vlákna

```
1 #include <pthread.h>
2 .
3 void *start_routine(void * dummy) {
4     printf("Thread: Hello.\n");
5     sleep(60);
6     return(NULL);
7 }
8
9 int main ( int argc, char * argv[] ) {
10     void *dummy;
11     pthread_t tid1, tid2;
12     .
13     /* Creating threads */
14     pthread_create(&tid1, NULL, &start_routine, NULL);
15     pthread_create(&tid2, NULL, &start_routine, NULL);
16
17     /* Waiting for threads */
18     pthread_join(tid1, &dummy);
19     pthread_join(tid2, &dummy);
20     .
21
22     return 0;
}
```

Příklad: POSIX vlákna

```
pthread_create(pthread_t *tid, ...,
              void *(*start_routine) (void *), ...)
```

- Funkce v aktuálním procesu vytvoří nové vlákno, které bude reprezentováno funkcí `start_routine()` a na adresu definovanou prvním parametrem `tid` uloží číslo nového vlákna.

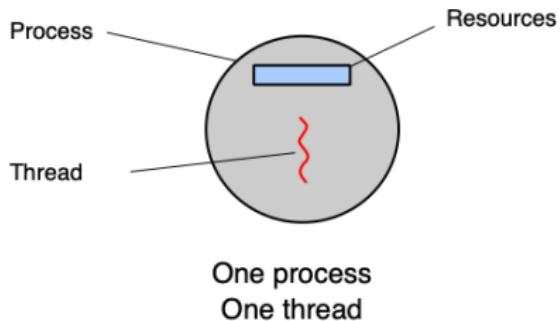
```
pthread_join(pthread_t tid, ...)
```

- Vlákno, ze kterého tuto funkci zavoláme, bude čekat na dokončení vlákna, které je se specifikováno prvním parametrem `tid`.

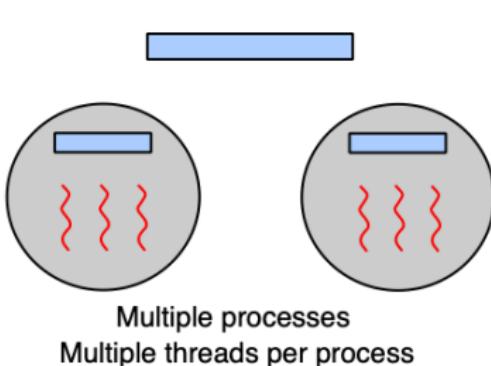
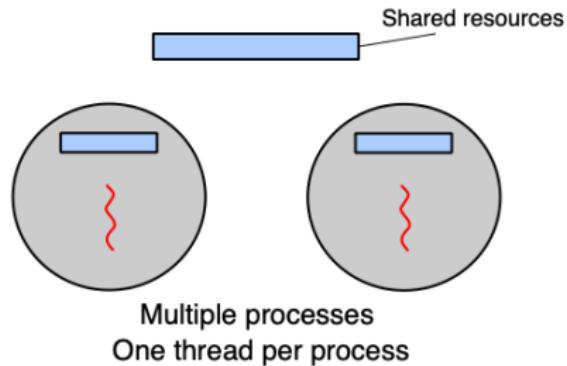
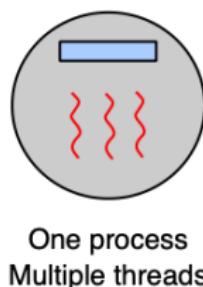
Bližší informace viz. manuálové stránky: `man pthread_create` a `man pthread_join`.

Multitasking/Multithreading

Process-based multitasking



Thread-based multitasking Multithreading



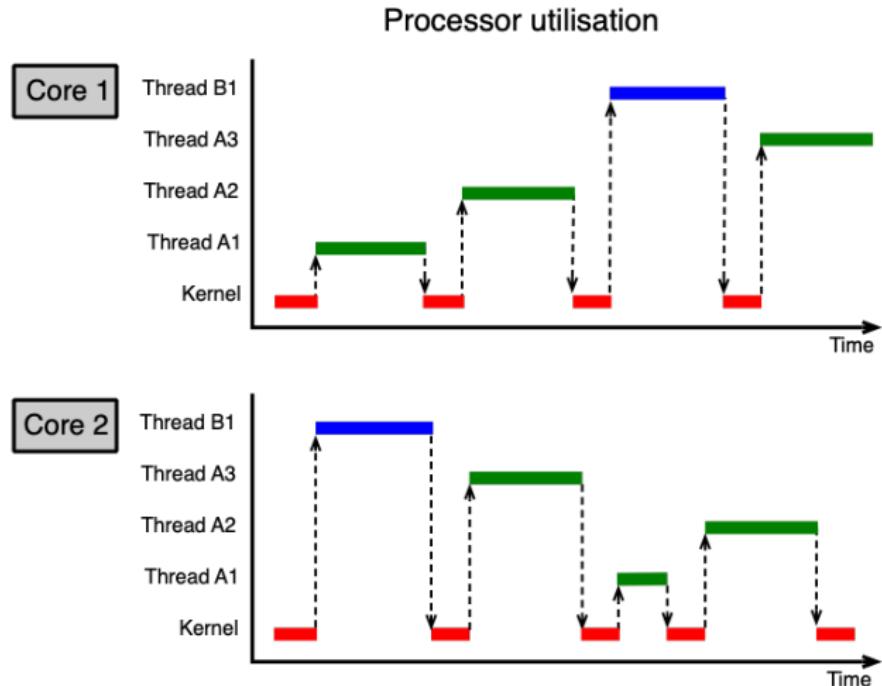
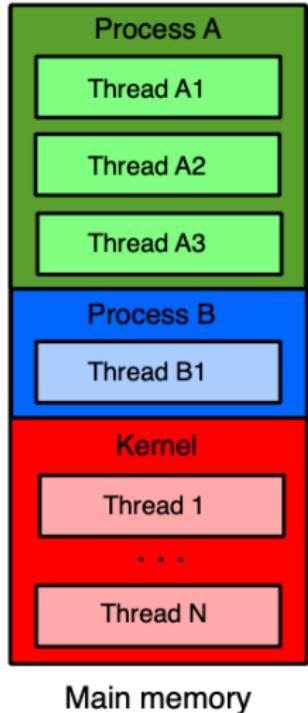
- Jádro OS a vytvořená vlákna sdílí omezený počet jader výpočetního systému.
- **Jedno vlákno (instrukční proud) může být v jednom okamžiku zpracováváno jedním "logickým" jádrem CPU.**
- Aby se vlákna "rozumným" způsobem podělila o omezený počet jader, tak se vlákna na jádrech střídají nejčastěji pomocí preemptivního plánování.
- **Preemptivní plánování vláken**

- ▶ Vláknu je přiděleno volné jádro CPU, pokud ho jádro OS vybere na základě plánovacích kriterií (např. priority,...).
- ▶ Jádro OS vláknu přidělí **časové kvantum**, během kterého vlákno bude zpracováváno jádrem CPU.
- ▶ Vláknu je jádro CPU odebráno pokud
 - ★ dojde k uplynutí časového kvanta (přerušení od časovače),
 - ★ vlákno provede systémové volání (např. V/V operaci),
 - ★ dojde k přerušení (např. je dokončena V/V operace,...).

• Přepínání kontextu

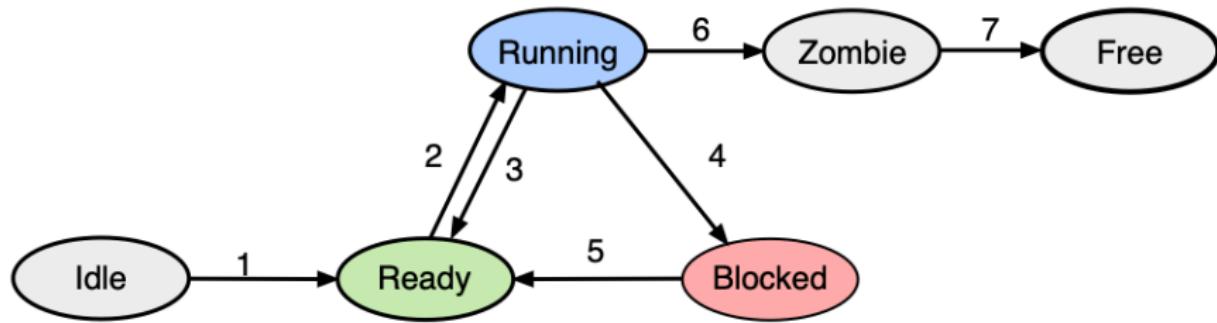
- ▶ Je to mechanismus, při kterém se vlákna vystřídají v používání jádra CPU.
- ▶ Kontextem se rozumí všechny nezbytné informace, které jsou nutné pro pozdější spuštění přerušeného vlákna od okamžiku přerušení (např. čítač instrukcí, obsahy registrů,...).
- ▶ Samotná vlákna žijí v iluzi, že běží bez přerušení od začátku do konce.
- ▶ Přepínání kontextu probíhá v několika krocích
 - 1 uloží se kontext původního vlákna,
 - 2 jádro OS naplánuje další vlákno,
 - 3 nastaví se kontext tohoto vlákna.

Přepínání kontextu



Stavy vláken

- Stav vlákna popisuje, co se s daným vláknem právě děje.
- Mezi základní stavy patří
 - ▶ **Idle**: vznik nového vlákna,
 - ▶ **Ready**: vlákno čeká až mu bude přiděleno jádro CPU,
 - ▶ **Running**: vlákno je zpracováváno jádrem CPU,
 - ▶ **Blocked**: vlákno čeká na událost (dokončení V/V operace, příchod signálu,...),
 - ▶ **Zombie**: vlákno je ukončováno, ale zatím ještě nebylo vše dokončeno,
 - ▶ **Free**: vlákno bylo kompletně zrušeno (pouze teoretický stav).

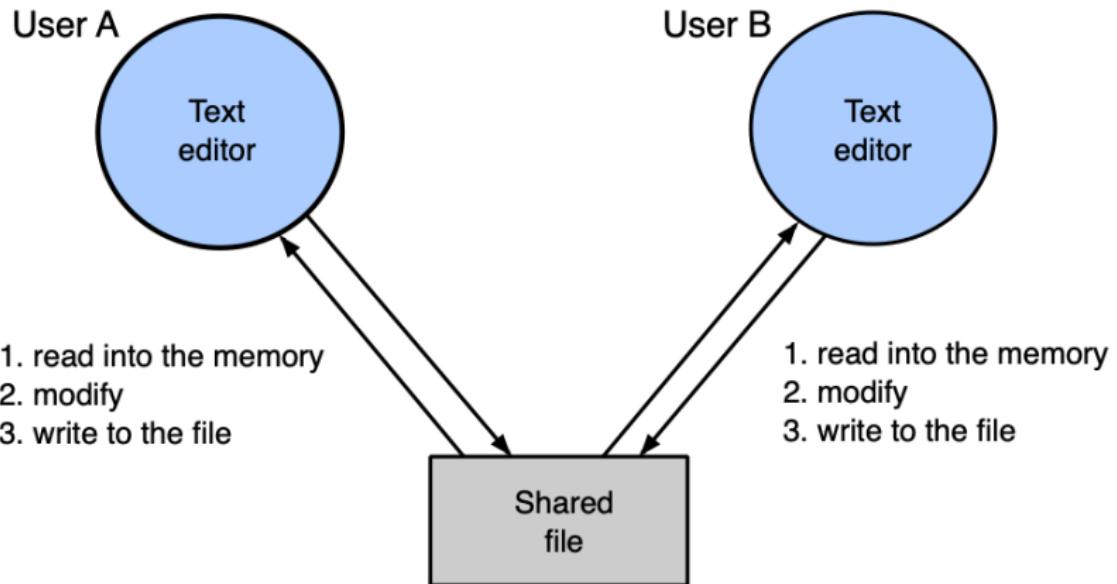


Časově závislé chyby (race conditions)

- Deterministický algoritmus
 - ▶ vždy ze stejných výchozích (vstupních) podmínek svým během vytvoří stejné výsledky.
- Časově závislé chyby
 - ▶ Situace, kdy dvě nebo několik vláken používá (čte/zapisuje) společné sdílené prostředky (např. sdílení proměnné, sdílené soubory,...) a výsledek deterministického algoritmu je závislý na rychlosti jednotlivých vláken, které používají tyto prostředky.
- Časově závislé chyby vykazují náhodný výskyt
⇒ špatně se detekují!
- Předcházet časově závislým chybám bychom měli správným návrhem paralelního algoritmu. Při hledání chyb nám mohou pomoci některé ladící nástroje (např. Valgrind,...), ale neměli bychom na ně stoprocentně spoléhat.

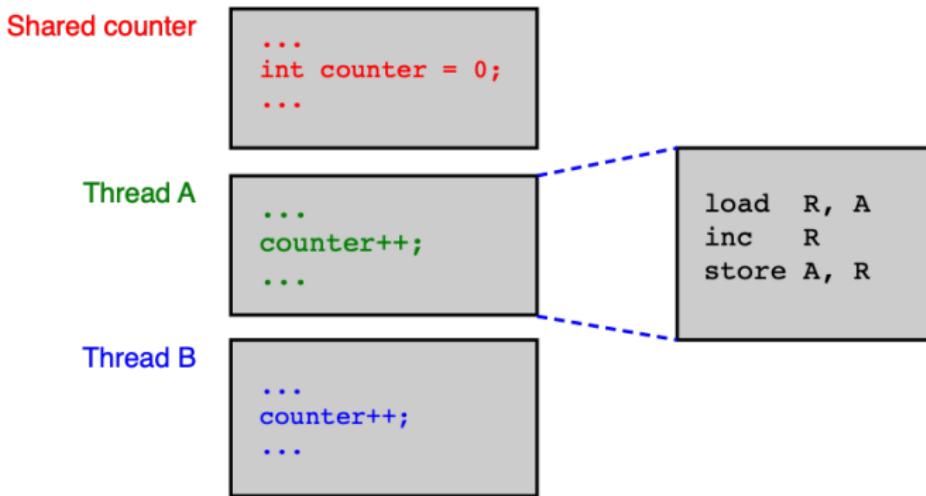
Příklad: Časově závislé chyby

- Dva uživatelé editují stejný soubor.



Příklad: Časově závislé chyby

- Dvě vlákna inkrementují sdílený čítač.



- Hodnota čítače je uložená v paměti na adresě A a procesor obsahuje registr R.
- Instrukce `load` načte hodnotu z paměti do registru.
- Instrukce `inc` inkrementuje hodnotu registru.
- Instrukce `store` uloží hodnotu z registru do paměti.

Kritické sekce (critical regions)

- Kritická sekce
 - ▶ Část programu, kde vlákna používají sdílené prostředky (např. sdílená proměnná, sdílený soubor, ...).
- Sdružené kritické sekce
 - ▶ Kritické sekce dvou (nebo více) vláken, které se týkají stejného sdíleného prostředku.
- Vzájemné vyloučení
 - ▶ Vlákňům není dovoleno sdílet stejný prostředek ve stejném čase.
⇒ Vlákna se nesmí nacházet ve stejné sdružené kritické sekci současně.

Korektní paralelní program

- Při psaní paralelních programů bychom měli dodržovat některá pravidla
 - ① Žádné předpoklady nesmí být kladený na rychlosť vláken a počet jader, která sdílí (různá vlákna mohou běžet různě rychle v závislosti na plánování vláken a zátěži systému).
 - ② Zajistit výlučný přístup ke sdíleným prostředkům
 - ★ Abychom toho dosáhli mohou být vlákna před kritickou sekcí pozastavena.
 - ★ Žádné vlákno ale nesmí čekat do nekonečna nebo "neúměrně dlouho" na vstup do kritické sekce.
 - ③ Mimo kritickou sekci by vlákno nemělo být blokováno (zomalováno) ostatními vlákny.

Použité zdroje

- ① A. S. Tanenbaum, H. Bos: *Modern Operating Systems (4th edition)*, Pearson, 2014.
- ② W. Stallings: *Operating Systems: Internals and Design Principles (9th edition)*, Pearson, 2017.
- ③ A. Silberschatz, P. B. Galvin, G. Gagne: *Operating System Concepts (9th edition)*, Wiley, 2012.

Operační systémy

Synchronizace vláken

Jan Trdlička



České vysoké učení technické v Praze, Fakulta informačních technologií
Katedra počítačových systémů

<https://courses.fit.cvut.cz/BI-OSY>

Obsah přednášky

- 1 Paralelní výpočet – problémy
- 2 Přehled synchronizačních nástrojů
- 3 Synchronizační problém: Kritická sekce
 - Aktivní čekání
 - Instrukce TSL
 - Vlastnosti
 - Inverzní prioritní problém
 - Blokující systémová volání/knihovní funkce
 - Zámky
 - Vlastnosti
- 4 Synchronizační problém: Producent-konzument
 - Podmíněné proměnné
 - Semafora
- 5 Synchronizační problém: Iterační výpočty
 - Blokující volání: Bariéry

• Bez použití synchronizace

- ▶ Časově závislé chyby (race conditions): situace, kdy více vláken používá (čte/zapisuje) společné sdílené prostředky během deterministického algoritmu a jeho výsledek je závislý na rychlosti vláken.

• Při použití synchronizace

- ▶ Uváznutí (deadlock): situace, kdy několik vláken čeká na událost, kterou může vyvolat pouze jedno z čekajících vláken.
- ▶ Livelock: situace, kdy několik vláken vykonává neužitečný výpočet (mění svůj stav), ale nemohou dokončit výpočet.
- ▶ Hladovění (starvation): situace, kdy je vlákno ve stavu "Ready" předbíháno a nedostane se po "dlouhou" dobu k prostředkům.

Přehled synchronizačních nástrojů

Hardware	SPARC v9: instrukce cas, ldstub, swap, ... x86-64: instrukce xchg, cmpxchg, ...
Jádro OS	Linux: atomic operation, spin locks, reader-writer locks, ... Solaris: spin locks, adaptive locks, reader-writer locks, ... MS Windows: executive dispatcher objects, slim reader-write locks, ...
Aplikace	Unix OS: pipes, signals, System V semaphores, message queues, ... MS Windows: mutexes, reader-writer locks, semaphores, events, ... POSIX Thread: mutexes, condition variables, semaphores, ... Languages: C++11, Java, ...

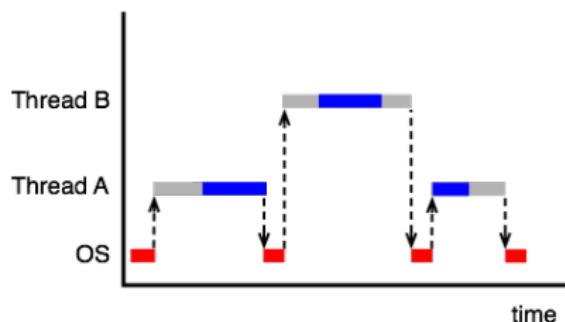
Synchronizace kritické sekce – aktivní čekání

- Pouze jedno vlákno může být uvnitř kritické sekce
⇒ ostatní vlákna musí počkat dokud se kritická sekce neuvolní.
- Toho lze docílit dvěma základními způsoby
 - ▶ pomocí aktivního čekání,
 - ▶ pomocí blokujících systémových volání/knihovnich funkcí.
- **Aktivní čekání (busy waiting, spinning)**
 - ▶ Sdílená proměnná indikuje obsazenost kritické sekce (zamčená/odemčená).
 - ▶ **Před vstupem do**
 - ★ **zamčené sekce**: vlákno ve smyčce "aktivně" testuje aktuální hodnotu proměnné do okamžiku než se sekce uvolní,
 - ★ **odemčené sekce**: vlákno změní hodnotu sdílené proměnné (zamkne kritickou sekci) a vstoupí do sekce.
 - ▶ **Po opuštění kritické sekce**
 - ★ Vlákno změní hodnotu sdílené proměnné (odemkne sekci).

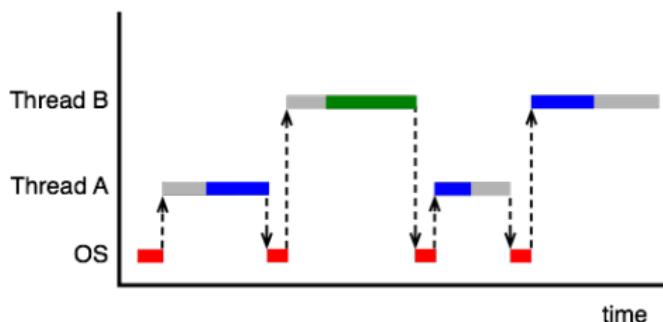
Synchronizace kritické sekce – aktivní čekání

- Předpokládejme, že dvě vlákna A a B přistupují ke společnému sdílenému prostředku.
- Následující obrázek ukazuje přístup bez synchronizace a synchronizaci pomocí aktivního čekání.

Race conditions



Busy waiting



- The thread is outside the critical region (it does not use the shared resource).
- The thread is inside the critical region (it uses the shared resource).
- The thread is outside the critical region and waits by busy waiting to acquire the resource.

Příklad: Aktivní čekání pomocí proměnné lock

- Vzájemné vyloučení pomocí sdílené proměnné `lock`, kterou vlákno nastaví když vstupuje do kritické sekce.
- Kritická sekce je
 - ▶ odemčená, pokud `lock` má hodnotu 0,
 - ▶ zamčená, pokud `lock` má hodnotu 1.

```
1 int lock = 0;                                     /* control access to critical region(CR) */
```

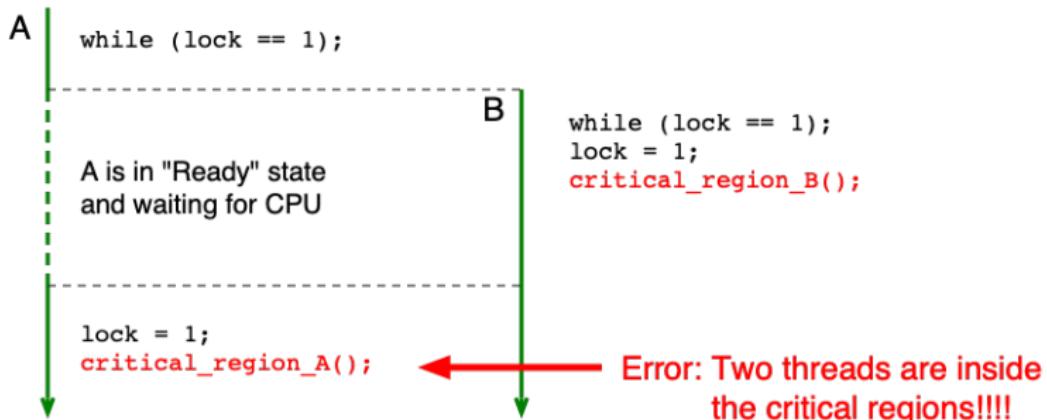
```
1 void thread_A(void)
2 {
3     while (TRUE)
4     {
5         ...
6         while ( lock == 1 ); /* busy waiting */
7         lock = 1;           /* enter CR */
8         critical_region_A();
9         lock = 0;;          /* leave CR */
10        ...
11    }
12 }
```

```
1 void thread_B(void)
2 {
3     while (TRUE)
4     {
5         ...
6         while ( lock == 1 ); /* busy waiting */
7         lock = 1;           /* enter CR */
8         critical_region_B();
9         lock = 0;;          /* leave CR */
10        ...
11    }
12 }
```

- Je to správné řešení?

Příklad: Aktivní čekání pomocí proměnné lock

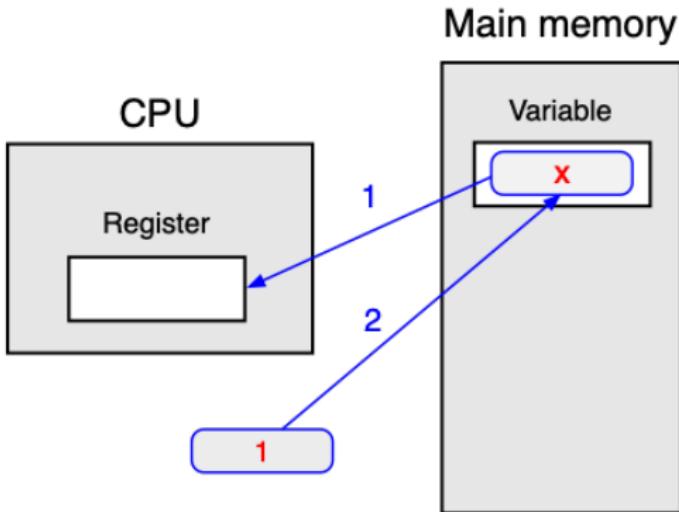
- Proč předchozí řešení je špatné?
- Předpokládejme, že v kritické sekci není žádné vlákno (lock má hodnotu 0).



- **Instrukce TSL (test-and-set-lock)**

- ▶ Hypotetická instrukce sloužící ke korektní implementaci aktivního čekání na hardwarové úrovni.
 - ▶ Samotná instrukce se skládá ze dvou kroků
 - 1 Načte obsah slova z dané adresy v paměti do registru.
 - 2 Nastaví obsah slova v paměti na nenulovou hodnotu (zamkne kritickou sekci).
 - ▶ Instrukce je atomická
 - ★ Žádné jiné vlákno během provádění této instrukce nemůže přistupovat ke slovu v paměti.
 - ★ Její implementace závisí na konkrétní hardwarové architektuře (např. zamčení paměťové sběrnice, ...).
- ⇒ korektní hardwarové řešení ve více-jádrových systémech se sdílenou pamětí.

Aktivní čekání: Instrukce TSL



The pair of operations 1 and 2 is executed atomically.

Aktivní čekání: Instrukce TSL

Průchod kritickou sekcí se skládá z následujících kroků.

- 1 Před vstupem do kritické sekce vlákno testuje, zda je sekce odemčená a pokud není, tak aktivně čeká.

```
enter_region:  
    tsl REGISTER, LOCK      | copy lock to register and set lock to 1  
    cmp REGISTER, #0         | was lock zero?  
    jne enter_region        | if it was nonzero, lock was set, so loop  
    ret                      | return to caller; critical region entered
```

- 2 V kritické sekci vlákno bezpečně používá sdílený prostředek.
- 3 Po upuštění kritické sekce vlákno musí sekci odemknout.

```
leave_region:  
    store LOCK, #0           | store a 0 in lock  
    ret                      | return to caller
```

- ISA reálných procesorů většinou neobsahuje přímo instrukci TSL, ale nabízí nějakou její variantu, pomocí které můžeme implementovat aktivní čekání (různé varianty aktivních zámků).
 - ▶ SPARC v9
 - ★ Load-store unsigned byte: `ldstub`,
 - ★ Compare and swap: `cas`,
 - ★ Swap register with memory: `swap`, ...
 - ▶ x86-64
 - ★ Exchange register / memory with register: `xchg`,
 - ★ Compare and exchange: `cmpxchg`, ...

Aktivní čekání: Vlastnosti

• Výhody

- ▶ Minimální režie, pokud vlákno nemusí čekat nebo čeká krátkou dobu před vstupem do kritické sekce.

• Nevýhody

- ▶ Vlákno, které čeká před vstupem do kritické sekce, zatíží jedno jádro CPU na 100%.
- ▶ V jistých situacích může aktivní čekání skončit uváznutím
⇒ inverzní prioritní problém.

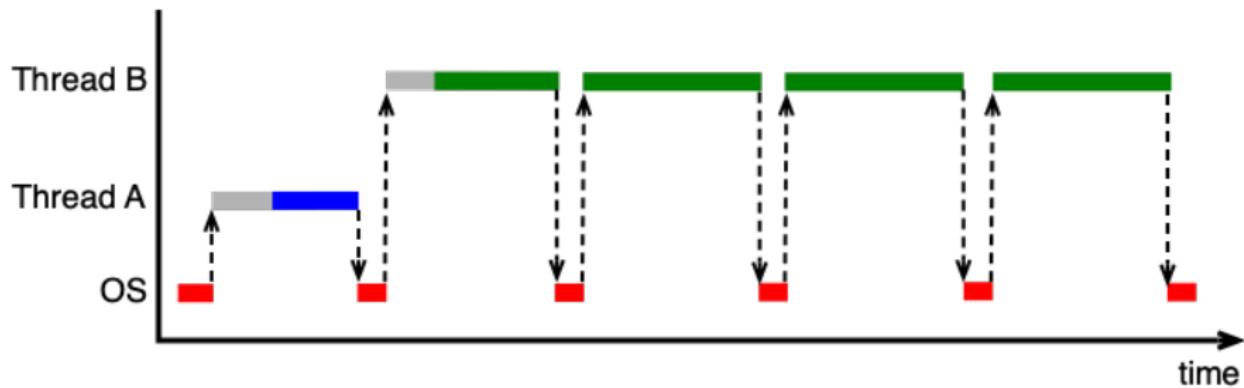
• Inverzní prioritní problém

- ▶ Nutné podmínky
 - ★ OS používá prioritní plánování vláken s fixní prioritou (priorita přiřazená vláknu se během jeho existence nemění).
 - ★ CPU má omezený počet n jader.
- ▶ Uváznutí nastane pokud
 - ★ vlákno A má nízkou prioritu a nachází se v kritické sekci,
 - ★ vlákno B má vyšší prioritu a čeká pomocí aktivního čekání na vstup do kritické sekce,
 - ★ všechna jádra v systému jsou obsazena vlákny s vyšší prioritou.

⇒ Aktivní čekání se vyplatí, pokud se očekává krátká doba čekání na prostředek a nehrozí inverzní prioritní problém (např. synchronizace v jádru OS).

Aktivní čekání: Inverzní prioritní problém

- 1) Assume two threads A and B and shared critical region.
- 2) System has only one CPU with one core and uses the scheduling with fix priority.
- 3) The thread B has higher priority than the thread A.



- The thread is outside the critical region (it does not use the shared resource).
- The thread is inside the critical region (it uses the shared resource).
- The thread is outside the critical region and waits by busy waiting to acquire the resource.

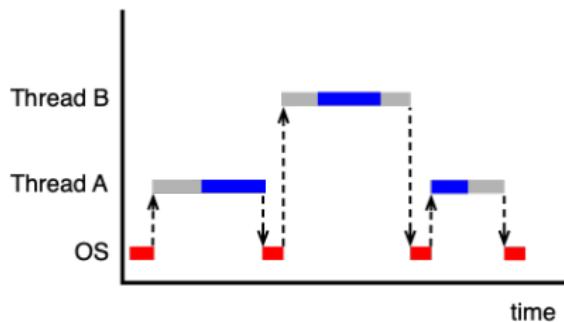
Synchronizace kritické sekce – blokující volání

- **Blokující systémové volání/knihovní funkce** je implementováno obvykle pomocí datových struktur, které umožňují
 - ▶ si pamatovat stav kritické sekce (odemčená/zamčená),
 - ▶ udržovat seznam vláken, která čekají na vstup do kritické sekce.
- **Před vstupem do kritické sekce**
 - ▶ **zamčená sekce:** vlákno provede systémové volání/knihovní funkci, které ho zablokuje (přepne jeho stav do stavu "Blocked"), a tím pádem vláknu přestane být přidělován procesor ⇒ **pasivně čeká** na uvolnění sekce,
 - ▶ **odemčená sekce:** vlákno provede systémové volání/knihovní funkci, které ho nezablokuje ale pouze si zapamatuje, že sekce je zamčená, a vlákno "vstoupí" do sekce.
- **Po opuštění kritické sekce**
 - ▶ vlákno pomocí systémového volání/knihovní funkce probudí čekající vlákno/vlákna,
 - ▶ v případě, že již žádná vlákna nečekají, pak si zapamatuje, že sekce je odemčená.

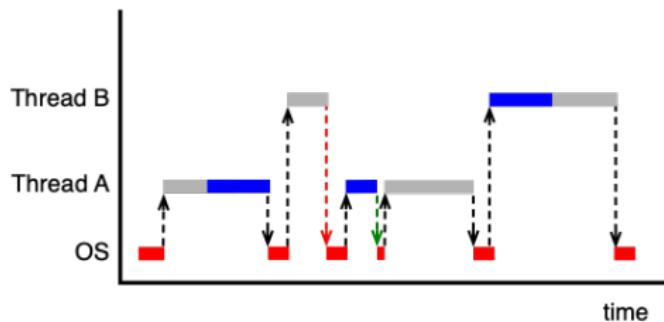
Synchronizace kritické sekce – blokující volání

- Předpokládejme, že dvě vlákna A a B přistupují ke společnému sdílenému prostředku.
- Následující obrázek ukazuje přístup bez synchronizace a synchronizaci založenou na blokování.

Race conditions



Blocking system call



- The thread is outside the critical region (it doesn't use the shared resource).
- The thread is inside the critical region (it uses the shared resource).
- Syscall/library function blocks the calling thread.
- Syscall/library function unblocks one waiting thread or unlock critical region.

Blokující volání: Zámky

- Zámek je často označován jako **mutex** (MUTual EXclusion lock) a pamatuje si
 - ▶ svůj stav (zamčený/odemčený),
 - ▶ kdo je jeho vlastníkem (vlákno, které ho zamklo),
 - ▶ množinu vláken, která jsou na něm blokovaná.
- Nad zámkem `mutex` jsou definovány atomické operace
 - ▶ `mutex_lock (mutex_t *mutex)`
 - ★ Pokud je `mutex` odemčený, tak ho zamkne. Volající vlákno se stane vlastníkem zámku.
 - ★ Pokud je `mutex` zamčený, tak zablokuje volající vlákno.
 - ▶ `mutex_unlock (mutex_t *mutex)`
 - ★ Měl by volat pouze vlastník zámku (jinak skončí chybou).
 - ★ Pokud jsou nějaká vlákna blokována zámkem, tak se jedno z nich probudí.
 - ★ Pokud již žádné vlákno není blokováno, tak se odemkne `mutex`.

• Příklady

- ▶ **POSIX:** datový typ `pthread_mutex_t` s funkcemi
`pthread_mutex_lock()`, `pthread_mutex_trylock`,
`pthread_mutex_unlock`, ...
- ▶ **C++:** třídy `std::mutex`, `std::lock_guard`, `std::unique_lock`

Blokující volání: Zámky

```
1 mutex_t mutex;                                /* control access to critical region(CR) */
```

```
1 void thread_A(void)
2 {
3     while (TRUE)
4     {
5         ...
6         mutex_lock(&mutex);          /* enter CR */
7         critical_region_A();
8         mutex_unlock(&mutex);      /* leave CR */
9         ...
10    }
11 }
```

```
1 void thread_B(void)
2 {
3     while (TRUE)
4     {
5         ...
6         mutex_lock(&mutex);          /* enter CR */
7         critical_region_B();
8         mutex_unlock(&mutex);      /* leave CR */
9         ...
10    }
11 }
```

Blokující volání: Vlastnosti

- **Výhody**

- ▶ Čekání na vstup do kritické sekce nepředstavuje žádnou režii.

- **Nevýhody**

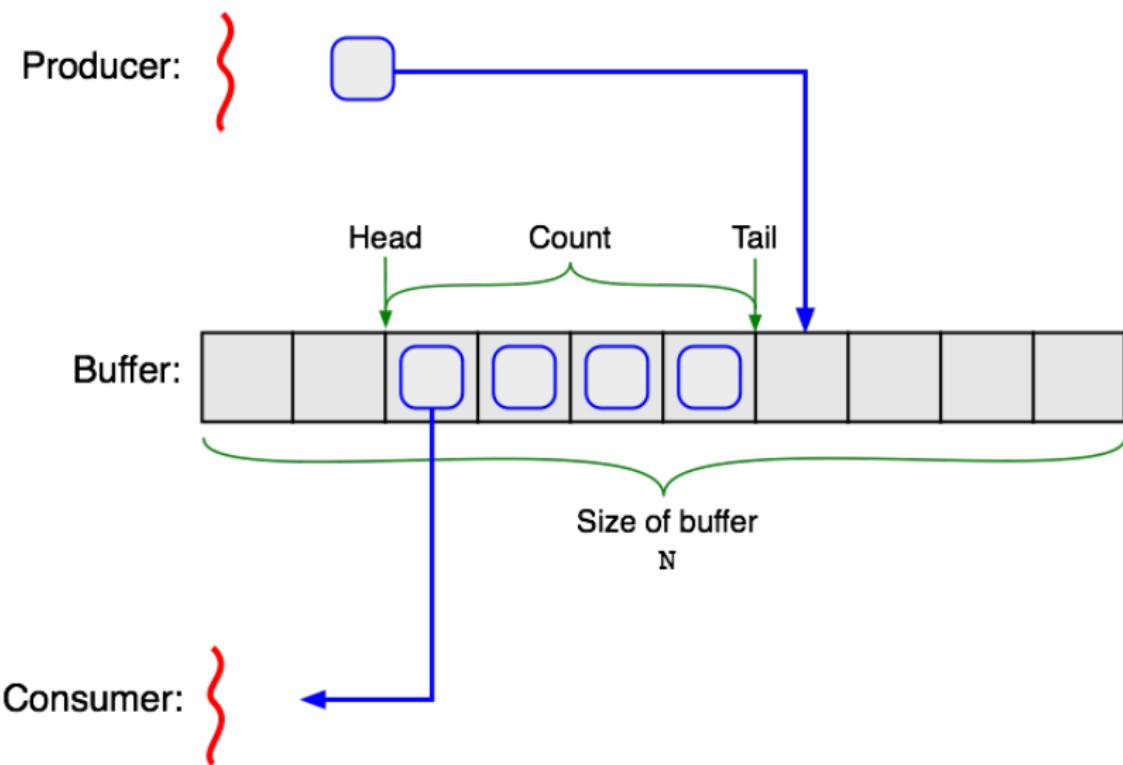
- ▶ Začátek a ukončení představují určitou režii, neboť je nutné změnit stav vlákna v jádře OS.

⇒ Vyplatí se, pokud se očekává delší doba čekání na prostředek.

Synchronizační problém: Producent-konzument

- Klasický synchronizační problém, který představuje situaci, kdy si několik vláken vyměňuje data prostřednictvím např. sdílené paměti s omezenou velikostí.
- **Producent**
 - ▶ produkuje data a vkládá je do sdílené paměti (fronty) s omezenou velikostí.
- **Konzument**
 - ▶ vybírá data ze sdílené paměti (fronty).
- **Problémy**
 - 1 Musíme zajistit výlučný přístup při vkládání/vybírání dat z fronty.
 - 2 Pokud je fronta prázdná \Rightarrow musíme zablokovat konzumenta.
 - 3 Pokud je fronta plná \Rightarrow musíme zablokovat producenta.

Synchronizační problém: Producent-konzument



Příklad: Blokující volání `wait()` a `signal()`

- Předpokládejme, že máme k dispozici následující hypotetické funkce
 - ▶ `wait()`
 - ★ Systémové volání, které zablokuje volající vlákno (nastaví jeho stav na "Blocked").
 - ▶ `signal(thread_t *thread)`
 - ★ Pokud je vlákno `thread` ve stavu "Blocked", tak se jeho stav změní na "Ready" (bude mu opět přidělováno CPU).
- V následujícím příkladě se pokusíme vyřešit problémy 2 a 3 v úloze producent-konzument pomocí těchto funkcí.
- Je řešení správné?

Příklad: Producents-konzument s wait() a signal()

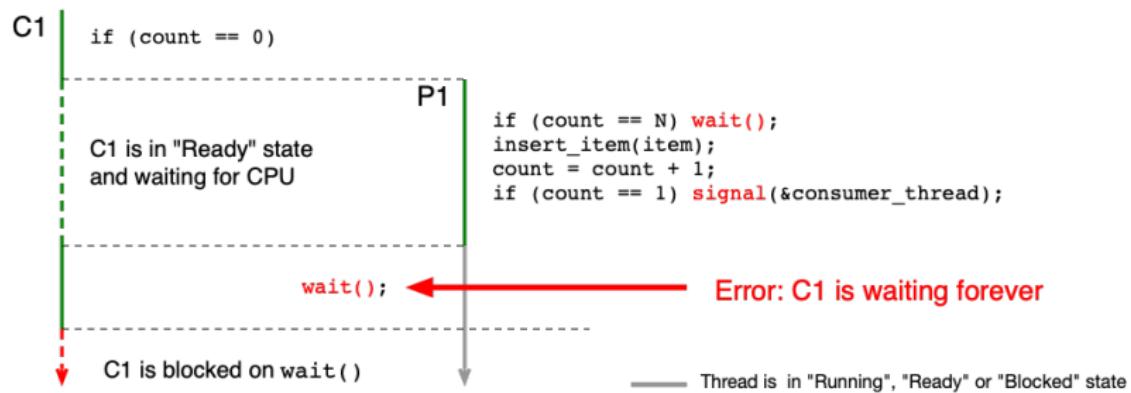
```
1 #define N 100                                /* number of slots in the buffer */
2 int      count = 0;                          /* number of items in buffer */
```

```
1 void producer(void) {
2     item_t item;
3     while (TRUE) {
4         item = produce_item();
5         if (count == N) wait();                /* if buffer is full, go to sleep */
6         insert_item(item);
7         count = count + 1;
8         if (count == 1) signal(&consumer_thread); /* was buffer empty? */
9     }
10 }
```

```
1 void consumer(void) {
2     item_t item;
3     while (TRUE) {
4         if (count == 0) wait();                /* if buffer is empty, go to sleep */
5         remove_item(&item)
6         count = count - 1;
7         if (count == N - 1) signal(&producer_thread); /* was buffer full? */
8         consume_item(&item);
9     }
10 }
```

Příklad: Producents-konzument s `wait()` a `signal()`

- Proč předchozí řešení je špatné?
- Předpokládejme, že fronta je prázdná (`count=0`) a existuje jeden producent a jeden konzument.



- Vyřešil by se problém pokud bychom se pokusili vždy
 - ▶ při vložení prvku probudit konzumenta,
 - ▶ při odebrání prvku probudit producenta?

Blokující volání: Podmíněné proměnné

- Podmíněná proměnná si pamatuje, která vlákna jsou na ní blokovány.
- Nad podmíněnou proměnnou `var` jsou typicky definovány operace
 - ▶ `cond_wait(cond_t *var, mutex_t *mutex)`
 - ★ Funkce musí být volána se zámkem `mutex`, který je zamčený volajícím vláknem.
 - ★ Funkce automaticky uvolní `mutex` a zablokuje volající vlákno, dokud nebude proměnná opět signalizována (předchozí signály nejsou ukládány).
 - ★ Po odblokování (návratu z funkce) je `mutex` opět zamčen.
 - ▶ `cond_signal(cond_t *var)`
 - ★ Odblokuje aspoň jedno ze zablokovaných vláken.
- **Příklady**
 - ▶ **POSIX:** datový typ `pthread_cond_t` s funkcemi `pthread_cond_wait()`, `pthread_cond_signal()`, `pthread_cond_broadcast()`, ...
 - ▶ **C++:** třída `std::condition_variable`.

Producents-konzument: Podmíněné proměnné

```
1 #define N 100                                /* number of slots in the buffer */
2 int count = 0;                               /* number of items in buffer */
3 mutex_t mutex;                            /* mutex */
4 cond_t cv_empty, cv_full;                  /* condition variables */
```

```
1 void producer(void)    {
2     item_t item;
3     while (TRUE)    {
4         item = produce_item();
5         mutex_lock(&mutex);
6         while (count == N) cond_wait(&cv_full, &mutex); /* if buffer is full, go to sleep */
7         insert_item(item);
8         count = count + 1;
9         cond_signal(&cv_empty);                         /* try to wakeup consumer */
10        mutex_unlock(&mutex);
11    }}
```

```
1 void consumer(void)    {
2     item_t item;
3     while (TRUE)    {
4         mutex_lock(&mutex);
5         while (count == 0) cond_wait(&cv_empty, &mutex); /* if buffer is empty, go to sleep */
6         remove_item(&item)
7         count = count - 1;
8         cond_signal(&cv_full);                          /* try to wakeup producer*/
9         mutex_unlock(&mutex);
10        consume_item(&item);
11    }}
```

Producent-konzument: Podmíněné proměnné

- Proč v předchozím řešení používáme cyklus while místo if?

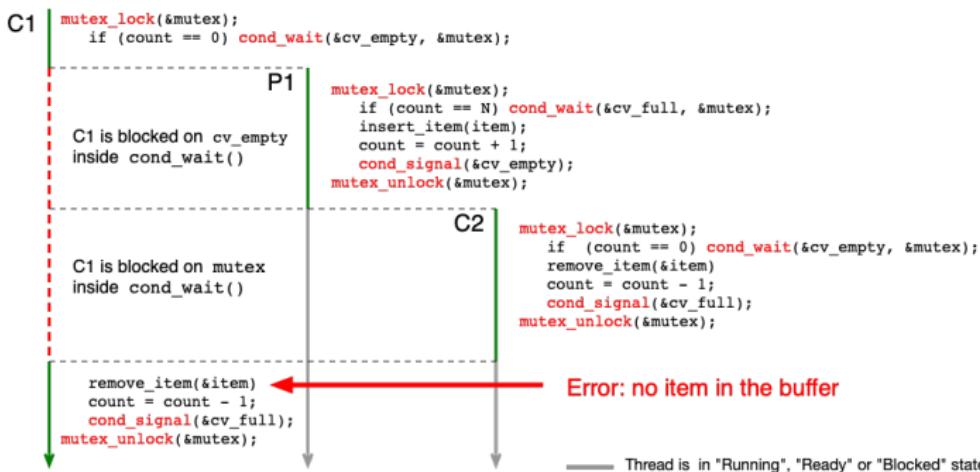
```
1 void producer(void)    {
2     item_t item;
3     while (TRUE)    {
4         item = produce_item();
5         mutex_lock(&mutex);
6         if (count == N) cond_wait(&cv_full, &mutex); /* if buffer is full, go to sleep */
7         insert_item(item);
8         count = count + 1;
9         cond_signal(&cv_empty);                      /* try to wakeup consumer */
10        mutex_unlock(&mutex);
11    }}
```

```
1 void consumer(void)    {
2     item_t item;
3     while (TRUE)    {
4         mutex_lock(&mutex);
5         if (count == 0) cond_wait(&cv_empty, &mutex); /* if buffer is empty, go to sleep */
6         remove_item(&item)
7         count = count - 1;
8         cond_signal(&cv_full);                      /* try to wakeup producer*/
9         mutex_unlock(&mutex);
10        consume_item(&item);
11    }}
```

Producent-konzument: Podmíněné proměnné

• Proč v předchozím řešení používáme cyklus while místo if?

- ▶ Předpokládejme, že fronta je prázdná a existuje více producentů a konzumentů.



- ▶ Některé API (POSIX Threads, WinAPI,...) umožňují falešné probuzení (spurious wakeup) uspaného vlákna, aniž by jiné vlákno zavolalo odpovídající funkci `cond_signal()`.

Blokující volání: Semaforu

- Datový typ semafor
 - ▶ obsahuje celočíselný čítač,
 - ▶ pamatuje si množinu vláken, která jsou na něm zablokována.
- Nad semaforem `sem` jsou definovány atomické operace
 - ▶ `sem_init(sem_t *sem, unsigned value)`
 - ★ Nastaví čítač na hodnotu `value` a vyprázdní frontu čekajících vláken.
 - ▶ `sem_wait(sem_t *sem)`
 - ★ Pokud je čítač větší než nula, potom se sníží o jedničku.
 - ★ V opačném případě se volající vlákno zablokuje a uloží do fronty.
 - ▶ `sem_post(sem_t *sem)`
 - ★ Pokud nějaká vlákna čekají ve frontě, potom se jedno z nich probudí.
 - ★ V opačném případě se čítač zvětší o jedničku.
- Příklady
 - ▶ Unix System V: `semget()`, `semctl()`, `semop()`.
 - ▶ POSIX: datový typ `sem_t` s funkcemi `sem_init()`, `sem_wait()`, `sem_post()`, ...
 - ▶ C++: nejsou implementovány.

Synchronizace kritické sekce – semafory

```
1 sem_t bin_sem;                                /* control access to critical region(CR) */
2
3 sem_init(&bin_sem, 1);
```

```
1 void Thread_A(void)
2 {
3     while (TRUE)
4     {
5         ...
6         sem_wait(&bin_sem);      /* enter CR */
7         critical_region_A();
8         sem_post(&bin_sem);    /* leave CR */
9         ...
10    }
11 }
```

```
1 void Thread_B(void)
2 {
3     while (TRUE)
4     {
5         ...
6         sem_wait(&bin_sem);      /* enter CR */
7         critical_region_B();
8         sem_post(&bin_sem);    /* leave CR */
9         ...
10    }
11 }
```

Synchronizace producent-konzument – semafory

```
1 #define N 100                                /* number slots in buffer */
2
3 mutex_t mutex;                               /* guards critical region (CR) */
4 sem_t empty;                                 /* counts empty slots */
5 sem_t full;                                  /* counts full slots */
6
7 sem_init(&empty, N);
8 sem_init(&full, 0);
```

```
1 void producer(void)
2 {
3     item_t item;
4     while (TRUE)
5     {
6         item = produce_item();
7         sem_wait(&empty);
8         mutex_lock(&mutex);    /* enter CR*/
9         enter_item(item);
10        mutex_unlock(&mutex); /* leave CR*/
11        sem_post(&full);
12    }
13 }
```

```
1 void consumer(void)
2 {
3     item_t item;
4     while (TRUE)
5     {
6         sem_wait(&full);
7         mutex_lock(&mutex);    /* enter CR*/
8         remove_item(&item);
9         mutex_unlock(&mutex); /* leave CR*/
10        sem_post(&empty);
11        consume_item(&item);
12    }
13 }
```

Blokující volání: Bariéry

- Bariéra umožňuje jednoduše synchronizovat iterační výpočty (např. výpočet n -té mocniny matice pomocí více vláken).
- Bariéra obsahuje
 - ▶ čítač, který definuje sílu bariéry (počet vláken, který bariéru prolomí),
 - ▶ frontu vláken, která jsou na bariéře blokována.
- Pro bariéru jsou definovány atomické operace
 - ▶ `barrier_init(barrier_t *bar, int value)`
 - ★ Funkce nastaví čítač bariéry na `value` a vyprázdní frontu čekajících vláken.
 - ▶ `barrier_wait(barrier_t *bar)`
 - ★ Pokud je čítač bariéry větší než 1, pak se čítač sníží o 1 a volající vlákno se zablokuje.
 - ★ Jinak se všechna blokovaná vlákna probudí a čítač se opět nastaví na `value`.

● Příklady

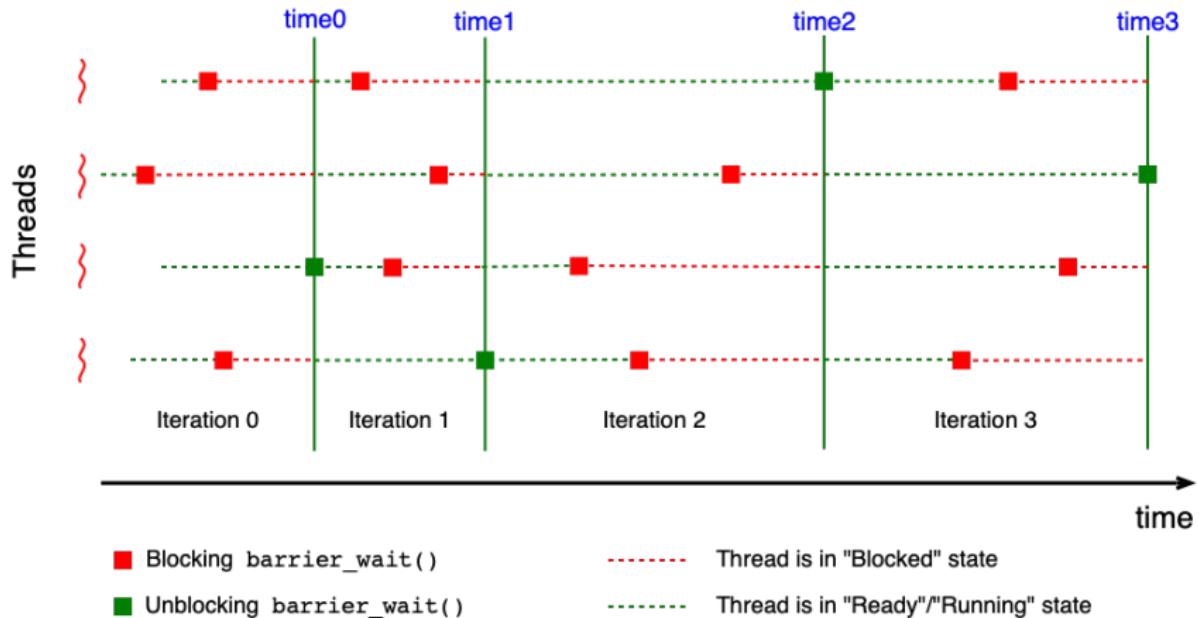
- ▶ **POSIX:** datový typ `sem_t` s funkcemi
`pthread_barrier_init()`, `pthread_barrier_wait()`, ...
- ▶ **C++:** třída `std::experimental::barrier`

Blokující volání: Bariéry

```
1 #define M 4                      /* number of threads */
2 #define N 10                     /* number of iterations */
3
4 barrier_t bar;
5 barrier_init(&bar, M);
```

```
1 void thread_function(void)
2 {
3     int i;
4     for ( i = 0; i < N; i++ )
5     {
6         iteration(i);
7         barrier_wait(&bar);
8     }
9 }
```

Blokující volání: Bariéry



- ① A. S. Tanenbaum, H. Bos: *Modern Operating Systems (4th edition)*, Pearson, 2014.
- ② W. Stallings: *Operating Systems: Internals and Design Principles (9th edition)*, Pearson, 2017.
- ③ A. Silberschatz, P. B. Galvin, G. Gagne: *Operating System Concepts (9th edition)*, Wiley, 2012.
- ④ R. Marejka: *Atomic SPARC: Using the SPARC Atomic Instructions*, Oracle Doc, 2008.

Operační systémy

Klasické synchronizační úlohy

Jan Trdlička



České vysoké učení technické v Praze, Fakulta informačních technologií
Katedra počítačových systémů

<https://courses.fit.cvut.cz/BI-OSY>

Obsah přednášky

1 Večeřící filosofové

- Definice
- Naivní řešení
- Správné řešení
- Správné optimální řešení
- Správné optimální řešení

2 Čtenáři-písáři

- Definice
- Správné řešení
- Problém s hladověním
- Optimální spravedlivé řešení

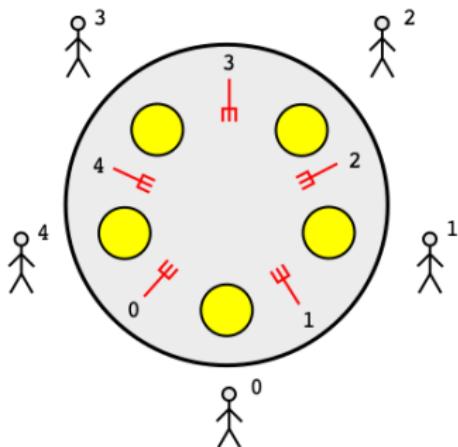
3 Spící holiči

- Definice
- Správné optimální řešení

Večeřící filosofové: Definice

- Klasický synchronizační problém, který reprezentuje situaci, kdy několik vláken soutěží o omezený počet prostředků.
- **Popis problému**
 - ▶ V systému je N filosofů, kteří sedí kolem kulatého stolu.
 - ▶ Před každým filosofem se nachází talíř s jídlem a mezi sousedními talíři je vždy jedna vidlička (celkem je N vidliček).
 - ▶ Pokud dostane filosof hlad, musí získat obě vidličky, které leží nalevo a napravo od jeho talíře.
 - ▶ Filosof se může nacházet ve třech stavech
 - ★ přemýslí (nepotřebuje žádné prostředky),
 - ★ má hlad a pokouší se získat vidličky (snaží se alokovat prostředky),
 - ★ jí (používá prostředky).
- **Správné řešení:** nebude docházet k časově závislým chybám, uváznutí, livelocku, hladovění,...
- **Optimální řešení:** v jeden okamžik může jíst až $\lfloor N/2 \rfloor$ filosofů.
- Existuje několik řešení tohoto problému.

Večeřící filosofové: Naivní řešení



Naivní řešení

```
#define N      5
#define LEFT   (i % N)
#define RIGHT ((i+1) % N)

void philosopher(int i){
    while (TRUE){
        think();
        take_fork(LEFT);
        take_fork(RIGHT);
        eat();
        put_fork(LEFT);
        put_fork(RIGHT);
    }
}
```

- Filosofové jsou simulováni vlákny, která vykonávají funkci `philosopher()`.
- Funkce `take_fork(i)` se pokusí získat vidličku i
 - ▶ vidlička je volná \Rightarrow filosof získá vidličku,
 - ▶ vidlička je používána \Rightarrow filosof začne čekat na vidličku.
- Funkce `put_fork(i)` uvolní vidličku i .
- Bude toto řešení fungovat?

Večeřící filosofové: Naivní řešení

- Naivní řešení

- ▶ Může selhat v situaci kdy všichni filosofové vezmou levou vidličku současně ⇒ potom budou čekat až se uvolní pravá vidlička ⇒ uváznutí(deadlock).

- Vylepšené řešení

- ▶ Pokud není pravá vidlička k dispozici, filosof vrátí již alokovanou levou vidličku zpět na stůl a pokus o jídlo zopakuje "později".
- ▶ Toto řešení může selhat v situaci kdy by všichni filosofové prováděli stejné kroky ve stejných okamžicích.
 - 1 Všichni filosofové vezmou levou vidličku.
 - 2 Uvolní levou vidličku, protože nemají k dispozici pravou vidličku.
 - 3 Tento postup budou opakovat po stejné době ⇒ livelock.
- ▶ Pravděpodobnost, že filosofové budou takto synchronně fungovat je malá, ale nelze vyloučit.

Večeřící filosofové: Správné řešení

- Na stůl s vidličkami se můžeme dívat jako na kritickou sekci
⇒ budeme synchronizovat známým způsobem.

```
1 mutex_t  mutex;
2
3 void philosopher(int i){
4     while (TRUE) {
5         think();
6         mutex_lock (&mutex)
7         take_fork(LEFT);
8         take_fork(RIGHT);
9         eat();
10        put_fork(LEFT);
11        put_fork(RIGHT);
12        mutex_unlock (&mutex)
13    }
14 }
```

- Řešení neobsahuje časově závislé chyby ani uváznutí ⇒ správné.
- Pouze jeden filozof může jíst i když by mohlo jíst současně více filosofů ⇒ není optimální.

Večeřící filosofové: Správné optimální řešení

- Řešení pomocí mutexu a N semaforů.
- Je toto řešení správné (bez uváznutí,...) a optimální (jí více filosofů současně)?

```
1 #define N      5
2 #define LEFT   ((N+i-1) % N)
3 #define RIGHT  ((i+1)    % N)
4 typedef enum {thinking, hungry, eating} state_t; /* state of philosopher */
5 state_t state[N];
6 mutex_t mutex;
7 sem_t s[N];
8 for ( i = 0; i < N; i++ ) { state[i] = thinking; sem_init(&s[i], 0); }
```

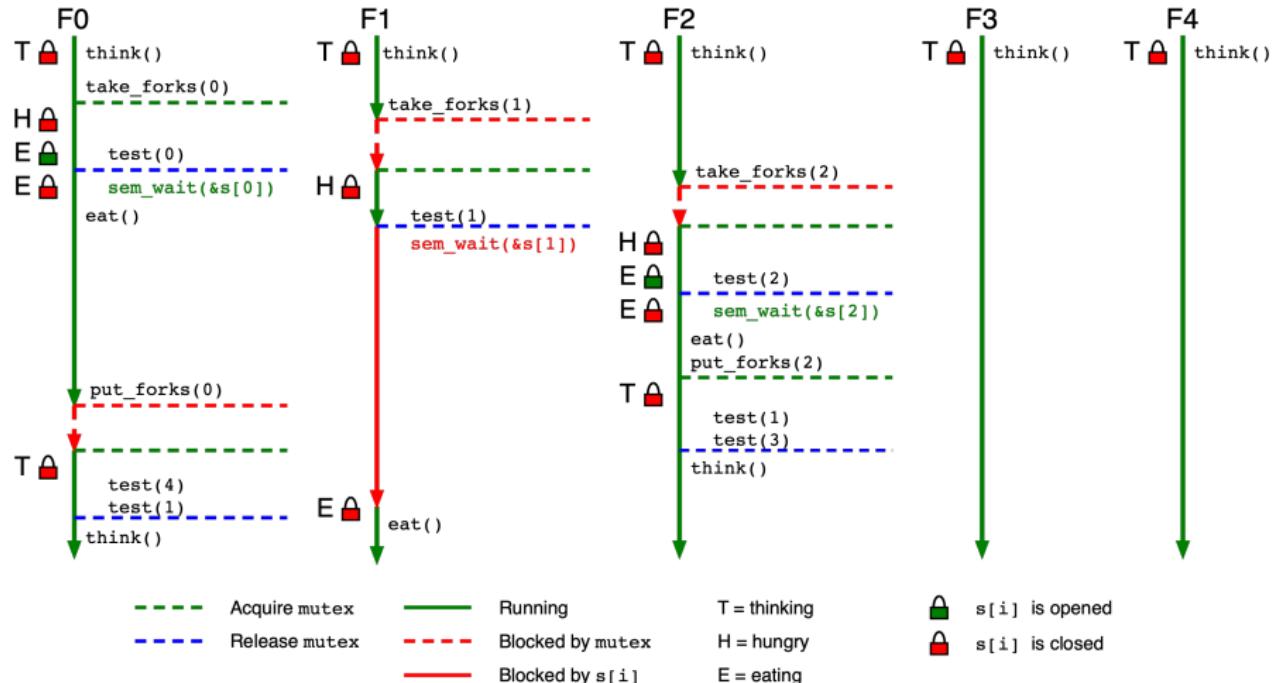
```
1 void philosopher(int i) {
2     while (TRUE) {
3         think();
4         take_forks(i);
5         eat();
6         put_forks(i);
7     }
8 }
```

```
1 void take_forks(int i) {
2     mutex_lock(&mutex);
3     state[i] = hungry;
4     test(i);
5     mutex_unlock(&mutex);
6     sem_wait(&s[i]);
7 }
```

```
1 void put_forks(int i) {
2     mutex_lock(&mutex);
3     state[i] = thinking;
4     test(LEFT);
5     test(RIGHT);
6     mutex_unlock(&mutex);
7 }
```

```
1 void test(int i) {
2     if (state[i] == hungry &&
3         state[LEFT] != eating &&
4         state[RIGHT] != eating)
5     {
6         state[i] = eating;
7         sem_post(&s[i]);
8     }
9 }
```

Večeřící filosofové: Správné optimální řešení



Večeřící filosofové: Správné optimální řešení

- Řešení pomocí mutexu a N podmíněných proměnných.
- Je toto řešení správné (bez uváznutí,...) a optimální (jí více filosofů současně)?

```
1 #define N      5
2 #define LEFT   ((N+i-1) % N)
3 #define RIGHT  ((i+1)    % N)
4 typedef enum {available, used} state_t;           /* state of fork */
5 state_t fork[N];
6 mutex_t mutex;
7 cond_t cv[N];
8 for ( i = 0; i < N; i++ ) { fork[i] = available; }
```

```
1 void philosopher(int i) {
2     while (TRUE) {
3         think();
4         take_forks(i);
5         eat();
6         put_forks(i);
7     }
8 }
```

```
1 void put_forks(int i) {
2     mutex_lock(&mutex);
3     fork[LEFT]  = available;
4     fork[RIGHT] = available;
5     cond_signal(&cv[LEFT]);
6     cond_signal(&cv[RIGHT]);
7     mutex_unlock(&mutex);
8 }
```

```
1 void take_forks(int i) {
2     mutex_lock(&mutex);
3     while ( ! forks_available(i) )
4         cond_wait(&cv[i], &mutex);
5     fork[LEFT]  = used;
6     fork[RIGHT] = used;
7     mutex_unlock(&mutex);
8 }
```

```
1 bool forks_available(int i) {
2     if ( fork[LEFT] == available &&
3         fork[RIGHT] == available )
4     {
5         return true;
6     }
7     return false;
8 }
```



Čtenáři-písáři: Definice

- Klasický synchronizační problém, ve kterém dva typy vláken soutěží o přístup ke společnému prostředku.
- **Popis problému**
 - ▶ V systému je jeden sdílený prostředek a dva typy vláken
 - ★ čtenáři R_0, \dots, R_{M-1} , kteří používají prostředek pouze pro čtení.
 - ★ písáři W_0, \dots, W_{N-1} , kteří mohou obsah prostředku modifikovat.
 - ▶ Pouze jeden písář může modifikovat obsah prostředku v jednom okamžiku.
 - ▶ **Optimální řešení:** Více čtenářů může číst současně pokud žádný písář nepřistupuje k prostředku.
 - ▶ **Spravedlivé řešení:** Pokud písář/čtenář čeká na sdílený prostředek, pak by ho žádný jiný čtenář ani písář neměl předběhnout.



Čtenáři-písáři: Správné řešení

- Čtenáři/písáři jsou simulováni vlákny, která vykonávají funkce `reader()`/`writer()`.
- Na sdílený prostředek se můžeme dívat jako na kritickou sekci
⇒ budeme synchronizovat známým způsobem.

```
1 mutex_t mutex;
```

```
1 void reader(void)
2 {
3     while (TRUE)
4     {
5         mutex_lock(&mutex);
6         read_data();
7         mutex_unlock(&mutex);
8         use_data();
9     }
10 }
```

```
1 void writer(void)
2 {
3     while (TRUE)
4     {
5         prepare_data();
6         mutex_lock(&mutex);
7         write_data();
8         mutex_unlock(&mutex);
9     }
10 }
```

- Řešení neobsahuje časově závislé chyby ani uváznutí ⇒ správné.
- Pouze jeden čtenář nebo pouze jeden písář může přistupovat ke sdílenému prostředku v jednom okamžiku. ⇒ není optimální

Čtenáři-písáři: Problém s hladověním

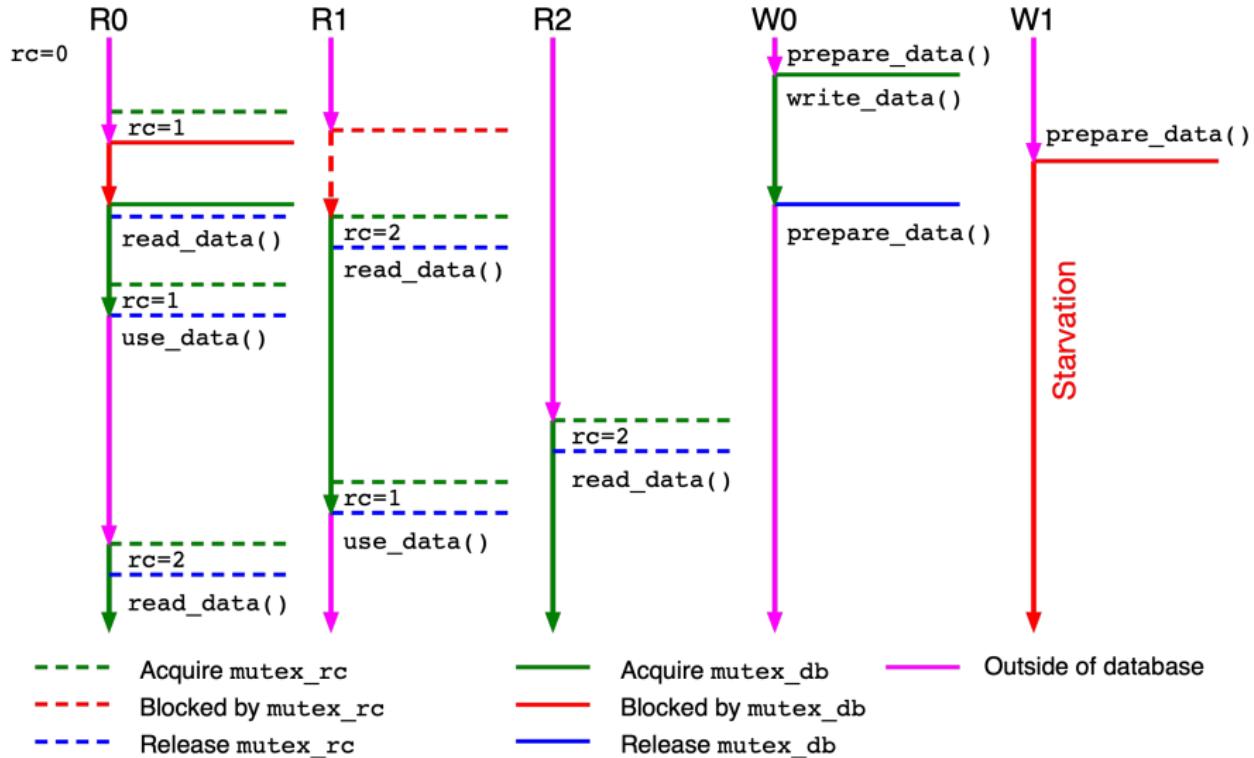
- Řešení pomocí dvou mutexů.
- Je to řešení
 - ▶ správné (bez časově závislých chyb, uváznutí,...),
 - ▶ optimální (více čtenářů může číst současně),
 - ▶ spravedlivé (čekající vlákna se nepředbíhají)?

```
1 int      rc = 0;          /* reader counter */
2 mutex_t  mutex_rc;       /* access to read counter */
3 mutex_t  mutex_db;       /* access to database */
```

```
1 void reader(void)
2 {
3     while (TRUE)
4     {
5         mutex_lock(&mutex_rc);
6         rc = rc + 1;
7         if (rc == 1) mutex_lock(&mutex_db);
8         mutex_unlock(&mutex_rc);
9         read_data();
10        mutex_lock(&mutex_rc);
11        rc = rc - 1;
12        if (rc == 0) mutex_unlock(&mutex_db);
13        mutex_unlock(&mutex_rc);
14        use_data();
15    }
16 }
```

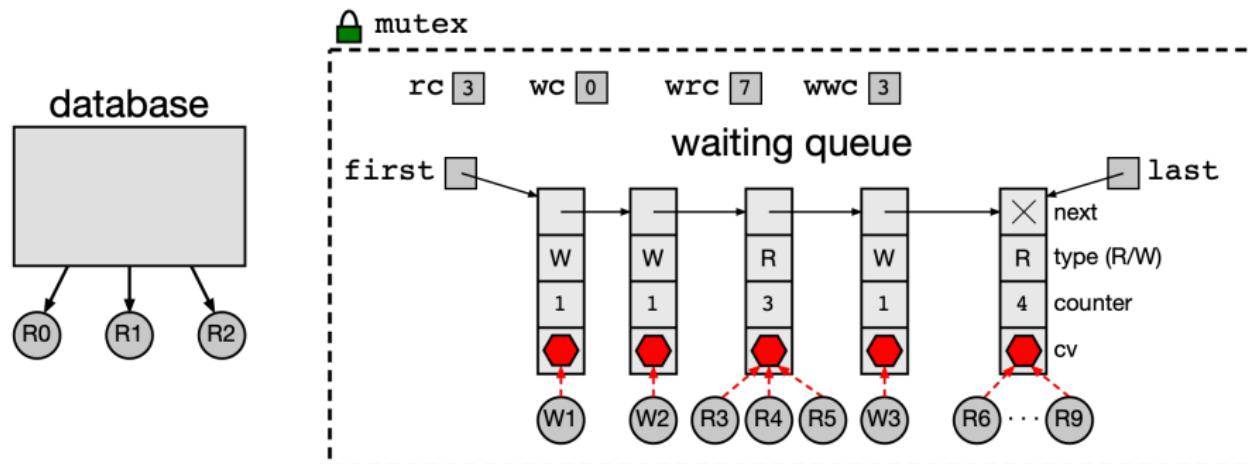
```
1 void writer(int i)
2 {
3     while (TRUE)
4     {
5         prepare_data();
6         mutex_lock(&mutex_db);
7         write_data();
8         mutex_unlock(&mutex_db);
9     }
10 }
```

Čtenáři-písáři: Problém s hladověním



- **Optimální řešení:** Více čtenářů může číst současně pokud žádný písař nepřistupuje k prostředku.
- **Spravedlivé řešení:** Pokud písař/čtenář čeká na sdílený prostředek, pak by ho žádný jiný čtenář ani písař neměl předběhnout.
 - ⇒ musíme si pamatovat v jakém pořadí čtenáři a písaři začínají čekat.
- Pokud vlákna jsou blokována na zámku, podmíněné proměnné nebo semaforu, tak většina API negarantuje jejich probuzení v pořadí FIFO.
- Jak to můžeme naimplementovat?

Čtenáři-písáři: Optimální spravedlivé řešení



- **mutex** chrání následující sdílené proměnné
 - ▶ **reader counter** rc = počet čtenářů, kteří právě čtou,
 - ▶ **writer counter** wc = počet písářů, kteří právě zapisují,
 - ▶ **waiting reader counter** wrc = počet čtenářů, kteří čekají na čtení,
 - ▶ **waiting writer counter** wwc = počet písářů, kteří čekají na zápis,
 - ▶ **waiting queue** = zřetězený seznam podmíněných proměnných, na kterých jsou blokováni čekající čtenáři/písáři.

Čtenáři-písáři: Optimální spravedlivé řešení

```
1 typedef enum {
2     writer,
3     reader
4 } type_t;
5
6 typedef struct {
7     item_t    *next;
8     type_t    type;
9     int        counter;
10    cond_t    cv;           /* condition variable */
11 } item_t;
12
13 mutex_t  mutex;
14 int      rc, wc, wrc, wwc;
15 item_t   *first, *last;
```

Čtenáři-písaři: Optimální spravedlivé řešení

- V programu budeme používat následující funkce

- ▶ `update_last_item()`

- ★ Pokud tuto funkci zavolá čtenář a v poslední položce fronty čekají čtenáři, tak čtenář inkrementuje `counter` poslední položky o 1 a uspí se na podmíněné proměnné poslední položky.
 - ★ Jinak vlákno vytvoří novou položku, nastaví `type`, `counter` položky na 1, přidá ji na konec fronty a uspí se na na podmíněné proměnné poslední položky.

- ▶ `update_first_item()`

- ★ Vlákno dekrementuje `counter` v první položce fronty o 1.
 - ★ Pokud je `counter` roven 0 (nikdo již zde nečeká), tak zruší první položku fronty.

- ▶ `wakeup_first_item()`

- ★ Postupně probudí čtenáře/písaře, kteří jsou uspaní na podmíněné proměnné v první položce fronty.
 - ★ Bud' můžeme v cyklu volat funkci `cond_signal()` nebo můžeme využít vlastností následujících reálných funkcí:
`Posix: pthread_cond_broadcast()`,
`C++: std::condition_variable::notify_all()`.

Čtenáři-písáři: Optimální spravedlivé řešení

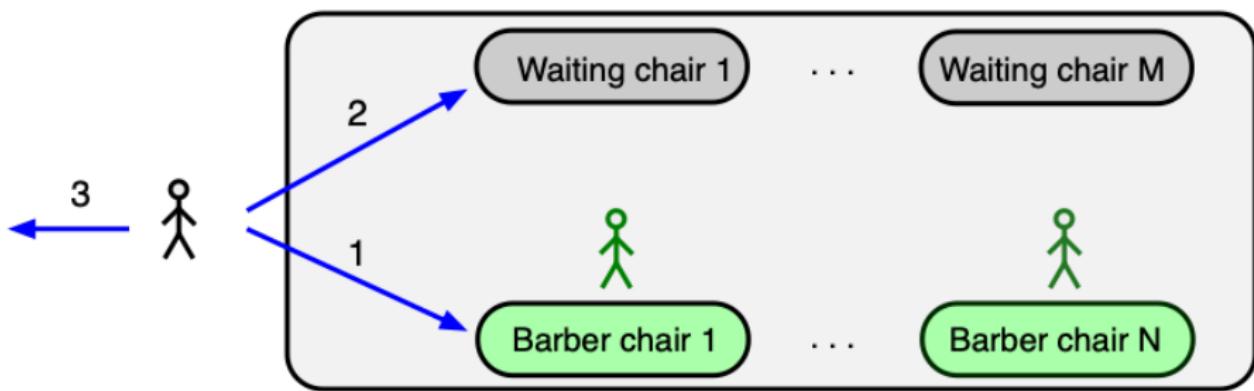
```
1 void reader(void) {
2     item_t *item;
3     type_t type = reader;
4
5     while (TRUE) {
6         mutex_lock(&mutex);
7         if ( wc > 0 || wwc > 0 ) { /* writer is there => go to wait */
8             wrc = wrc + 1;
9             item = update_last_item(last, type); /* update the last item of the queue */
10            while ( item != first || wc > 0 )
11                cond_wait(&item->cv, &mutex);
12                wrc = wrc - 1;
13                update_first_item(first); /* update the first item of the queue */
14            }
15            rc = rc + 1;
16            mutex_unlock(&mutex);
17
18            read_data();
19
20            mutex_lock(&mutex);
21            rc = rc - 1;
22            if ( rc == 0 ) wakeup_first_item(first); /* wake up writer in the first item */
23            mutex_unlock(&mutex);
24            use_data();
25        }
26    }
```

Čtenáři-písáři: Optimální spravedlivé řešení

```
1 void writer(void) {
2     item_t *item;
3     type_t type = writer;
4
5     while (TRUE) {
6         prepare_data();
7         mutex_lock(&mutex);
8         if ( rc > 0 || wc > 0 || wrc > 0 || wwc > 0 ) { /* anybody is there => go to wait */
9             wwc = wwc + 1;
10            item = update_last_item(last, type);           /* update the last item of the queue */
11            while ( item != first || rc > 0 || wc > 0 )
12                cond_wait(&item->cv, &mutex);
13            wwc = wwc - 1;
14            update_first_item(first);                   /* update the first item of the queue */
15        }
16        wc = wc + 1;
17        mutex_unlock(&mutex);
18
19        write_data();
20
21        mutex_lock(&mutex);
22        wc = wc - 1;
23        wakeup_first_item(first);           /* wake up writer/readers in the first item */
24        mutex_unlock(&mutex);
25    }
26 }
```

Spící holiči: Definice

- V holičství je N holičů (barbers), N křesel k holení (barber chairs) a M křesel k čekání (waiting chairs) pro zákazníky.
- Pokud není žádný zákazník v holičství, holič sedne do křesla k holení a usne.
- Pokud přijde zákazník, potom mohou nastat tři situace
 - 1 holič je volný (holič spí), tak ho probudí a nechá se ostříhat,
 - 2 holič není volný, ale je volné místo v čekárně, tak počká,
 - 3 jinak opustí holičství.



Spící holiči: Správné optimální řešení

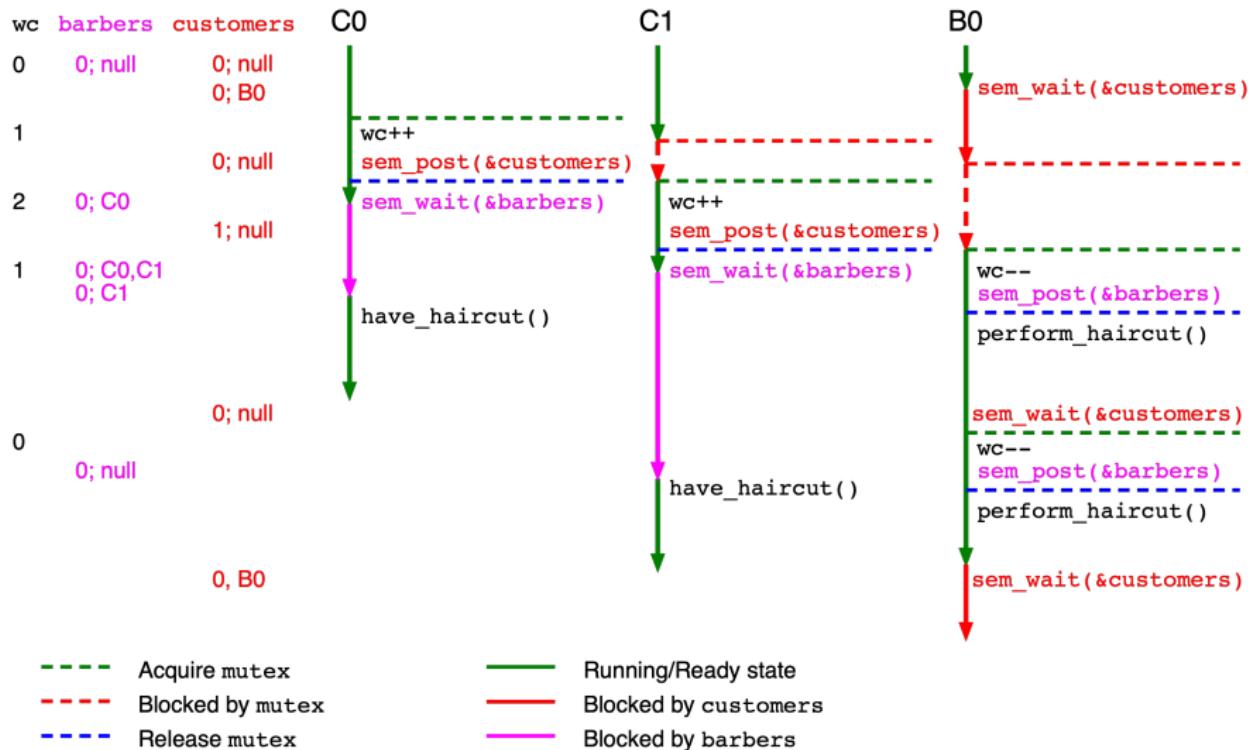
- Řešení pomocí jednoho mutexu a dvou semaforů.
- Zákazníci/holiči jsou simulováni vlákny, která vykonávají funkce `customer ()/barber ()`.
- Bude toto řešení
 - ▶ správné (neobsahuje časově závislé chyby, uváznutí,...),
 - ▶ optimální řešení (všichni holiči pracují, pokud je více zákazníků)?

```
1 #define M 5                                     /* waiting chairs          */
2 int wc = 0;                                    /* customers are waiting (not being cut) */
3 mutex_t mutex;                                /* for mutual exclusion      */
4 sem_t customers, barbers;                     /* # of customers waiting for service */
5 sem_init(customers, 0);                        /* # of barbers waiting for customers */
6 sem_init(barbers, 0);
```

```
1 void customer(void)
2 {
3     mutex_lock(&mutex);
4     if (wc < M)
5     {
6         wc = wc + 1;
7         sem_post(&customers);
8         mutex_unlock(&mutex);
9         sem_wait(&barbers);
10        have_haircut();
11    }
12    else {
13        mutex_unlock(&mutex);
14    }
15 }
```

```
1 void barber(void)
2 {
3     while (TRUE)
4     {
5         sem_wait(&customers);
6         mutex_lock(&mutex)
7         wc = wc - 1;
8         sem_post(&barbers);
9         mutex_unlock(&mutex);
10        perform_haircut();
11    }
12 }
```

Spící holiči: Správné optimální řešení



- ① A. S. Tanenbaum, H. Bos: *Modern Operating Systems (4th edition)*, Pearson, 2014.
- ② W. Stallings: *Operating Systems: Internals and Design Principles (9th edition)*, Pearson, 2017.
- ③ A. Silberschatz, P. B. Galvin, G. Gagne: *Operating System Concepts (9th edition)*, Wiley, 2012.

Operační systémy

Uváznutí vláken.

Jan Trdlička



České vysoké učení technické v Praze, Fakulta informačních technologií
Katedra počítačových systémů

<https://courses.fit.cvut.cz/BI-OSY>

Obsah přednášky

1 Výpočetní prostředky

- Alokace/uvolnění
- Alokační graf

2 Uváznutí

- Definice
- Coffmanovy podmínky

3 Strategie pro řešení uváznutí

- Pštrosí strategie
- Prevence uváznutí
- Předcházení vzniku uváznutí
- Detekce a zotavení

Výpočetní prostředky

• Výpočetní prostředek

- ▶ fyzický (např. fyzická paměť, tiskárna,...),
- ▶ logický (např. proměnná, soubor, mutex,...).

• Sdílené výpočetní prostředky

- ▶ Pokud je prostředek sdílen (používán) více vlákny současně, mohou vznikat časově závislé chyby ⇒ **je nutné zajistit výlučný přístup k prostředku** (v jednom okamžiku může být používán pouze jedním vláknem).

• Paralelní přístup k více sdíleným prostředkům

- ▶ Vlákna velmi často potřebují přistupovat k více prostředkům současně.

• Typy sdílených prostředků

- ▶ **Odnímatelné (preemptable)**: již alokovaný prostředek může být vláknu odebrán bez rizika dalších problémů (např. odložení procesu z fyzické paměti na disk při nedostatku fyzické paměti).
- ▶ **Neodnímatelné (nonpreemptable)**: nemohou být odebrány bez rizika (např. tiskárna,...).
- ▶ Bohužel většina prostředků je neodnímatelná.

- **Sekvence kroků při použití sdíleného prostředku vláknem**

- 1 alokace,
- 2 použití,
- 3 uvolnění.

- **Alokace prostředku**

- ▶ Vlákno žádá a prostředek prostřednictvím alokační funkce.
- ▶ Pokud je **prostředek volný**, pak je přidělen danému vláknu.
- ▶ Pokud je **prostředek již alokovaný**, pak existuje několik scénářů:
 - a Vlákno bude **blokováno**, dokud prostředek nebude k dispozici (např. `mutex_lock()`, `cond_wait()`, `sam_wait()`,...).
 - b Vlákno bude **blokováno maximálně po určitý čas** (např. `mutex_timedlock()`,...).
 - c Vlákno **nebude blokováno** (např. `fork()`, `malloc()`,...).
- ▶ V případech b a c vlákno zjistí např. na základě návratové hodnoty funkce, zda mu byl prostředek přidělen, či nikoliv. Vlákno potom musí rozhodnout, jak bude výpočet dál pokračovat.

• Uvolnění prostředku

- ▶ Vlákna by sama měla uvolňovat alokované prostředky!
- ▶ Některé prostředky uvolňuje jádro OS automaticky v okamžiku zániku procesu (např. paměť alokovaná pomocí `malloc()`).
- ▶ Jiné prostředky se neuvolní ani po zániku procesu (např. sdílená paměť alokovaná pomocí `shmget()`).

• V tomto textu budeme vycházet z následujících předpokladů.

- ▶ Pokud nejsou prostředky volné, potom vlákna budou čekat dokud se požadované prostředky neuvolní.
- ▶ Alokované prostředky budou používány vlákny po konečnou dobu a potom je vlákna sama uvolnění.

• Uváznutí (deadlock)

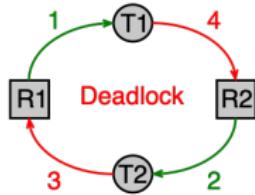
- ▶ Situace, kdy několik vláken čeká na událost/prostředek, kterou může vyvolat/uvolnit pouze jedno z čekajících vláken.

Alokační graf

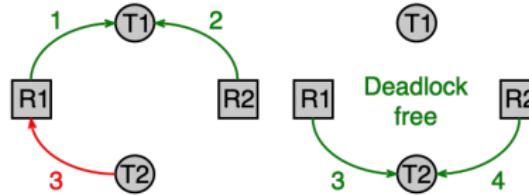
- Znázorňuje alokaci prostředků jednotlivými vlákny.
- Je to orientovaný graf s dvěma typy uzelů (prostředky/vlákna).
- Hrany grafu mají orientaci
 - ▶ od prostředku k vláknu, pokud vlákno má prostředek již alokovaný,
 - ▶ od vlákna k prostředku, pokud vlákno čeká na daný prostředek.
- Číslo hrany může reprezentovat pořadí alokace.

(T) Thread
(R) Resource

The thread T is waiting for the resource R.
The thread T is holding the resource R.



Order of allocation I



Order of allocation II

- Každá smyčka v grafu představuje uváznutí (vlákna ve smyčce čekají a nemohou pokračovat).
- Uváznutí závisí na pořadí v jakém jsou prostředky alokovány.

Coffmanovy podmínky

- Uváznutí nastane pouze pokud jsou **splněny následující podmínky.**
 - ① **Vzájemné vyloučení:** každý prostředek je buď přidělen právě jednomu vláknu a nebo je volný (prostředek nemůže být sdílen více vlákn).
 - ② **Podmínka neodnímatelnosti:** prostředek, který byl již přidělen nějakému vláknu, nemůže mu být násilím odebrán (musí být dobrovolně uvolněn daným vláknem).
 - ③ **Podmínka "drž a čekej":** vlákno, které má již přiděleny nějaké prostředky, může žádat o další prostředky (vlákno může žádat o prostředky postupně).
 - ④ **Podmínka kruhového čekání:** musí existovat smyčka dvou nebo více vláken, ve které každé vlákno čeká na prostředek přidělený dalšímu vláknu ve smyčce.
- První tři podmínky jsou nutné ale ne dostačující \Rightarrow k uváznutí může dojít. Poslední podmínka představuje samotné uváznutí.
- Pokud aspoň jedna z podmínek není splněna, nemůže dojít k uváznutí.

Strategie pro řešení uváznutí

① Pštrosí strategie

- ▶ Ignorování celého problému.

② Prevence uváznutí

- ▶ Pomocí nesplnění aspoň jedné z Coffmanových podmínek.

③ Předcházení vzniku uváznutí

- ▶ Na základě pečlivé alokace prostředků.

④ Detekce uváznutí a zotavení

- ▶ K uváznutí může dojít, ale je detekováno a odstraněno.

Pštrosí strategie

- Strategie, ve které se problém uváznutí neřeší/řeší částečně.
- Pokud k uváznutí dojde je vyžadován zásah uživatele/administrátora.
- **Tato strategie má opodstatnění za následujících podmínek**
 - ▶ systém obsahuje velký počet různě se chovajících vláken a velký počet různých prostředků,
 - ▶ pravděpodobnost výskytu uváznutí je relativně malá,
⇒ řešení uváznutí by bylo příliš drahé.
- Praktické řešení pro většinu univerzálních OS (MS Windows, OS unixového typu,...).
 - ▶ Jádro OS je navrženo jako "deadlock free".
 - ▶ Na úrovni procesů/vláken se problém řeší pouze částečně.
 - ★ Prostředky systému jsou omezené (fyzická paměť, systémy souborů, maximální počet vláken,...).
 - ★ Uživatelské vlákna se chovají nepredikovatelně.
 - ★ Částečně lze řešit různými limity (viz. unixový příkaz ulimit -a).
- **Toto řešení není přijatelné v "fault tolerant" systémech.**

Prevence uváznutí

- Tato strategie je založená na porušení aspoň jedné z Coffmanových podmínek ⇒ zamezíme vzniku uváznutí.

① Porušení podmínky "vzájemného vyloučení"

- ▶ Pokud je prostředek používán více vlákny pro čtení i zápis, pak v praxi tuto podmínsku nelze porušit bez rizika vzniku časově závislých chyb.

② Porušení podmínky "neodnímatelnosti prostředku"

- ▶ Tento přístup je vhodný pouze v případech, kdy je možné uložit (zapamatovat) si stav prostředku tak, aby později mohl být opět obnoven do původního stavu.
- ▶ Jako příklad může sloužit způsob sdílení jádra CPU více vlákny (přepínání kontextu).

③ Porušení podmínky "drž a čekej"

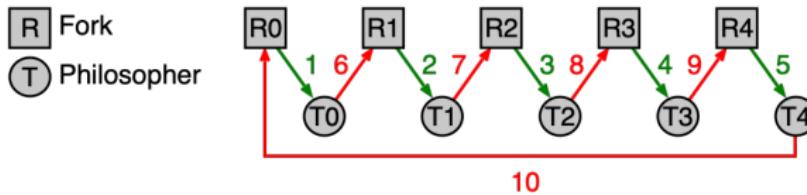
- ▶ Pokud má vlákno dopředu informaci o tom, které prostředky bude během své existence používat ⇒ lze všechny prostředky alokovat najednou v jednom kroku před jejich použitím (vlákno získá vše nebo začne čekat).
- ▶ Tento typ alokace většinou povede k horšímu využití prostředků.

④ Porušení podmínky "kruhového čekání"

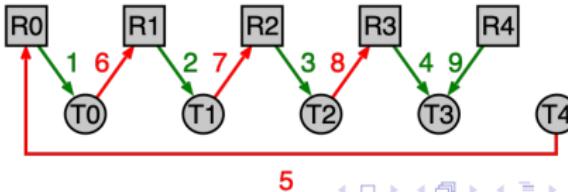
- ▶ Myšlenka porušení této podmínky je založená na vhodném číslování prostředků a jejich přidělování ve vzestupné pořadí.
- ▶ Každému prostředku přiřadíme jedinečné číslo v rámci systému
 - ★ množina prostředků $R = R_1, \dots, R_m$,
 - ★ číslování (mapování jedna k jedné) $F : R \rightarrow N$, kde N je množina celých čísel.
- ▶ Vlákno potom může žádat o libovolné prostředky, ale pouze v rostoucím číselném pořadí
 - ★ pokud vlákno žádá o prostředek R_j , pak pro každý prostředek již přidělený vláknu R_i musí platit $F(R_j) > F(R_i)$.
- ▶ Tímto způsobem přidělování nevznikne v alokačním grafu smyčka.
- ▶ Vhodné očíslování prostředků nemusí vždy existovat.
- ▶ Pomocí tohoto přístupu řešíme problém uváznutí ve fázi návrhu/kompilace programu.

Příklad: Prevence "kruhového čekání"

- Uvažujme "naivní řešení" večeřících filosofů
 - Prostředky (vidličky) se alokují pomocí funkce `take_fork(i)`.
 - Pokud vidlička není volná, tak tato funkce zablokuje filosofa, ze kterého byla zavolána.
 - Pokud všichni filosofové "současně" alokovali svou levou vidličku
⇒ **řešení s uváznutím**.



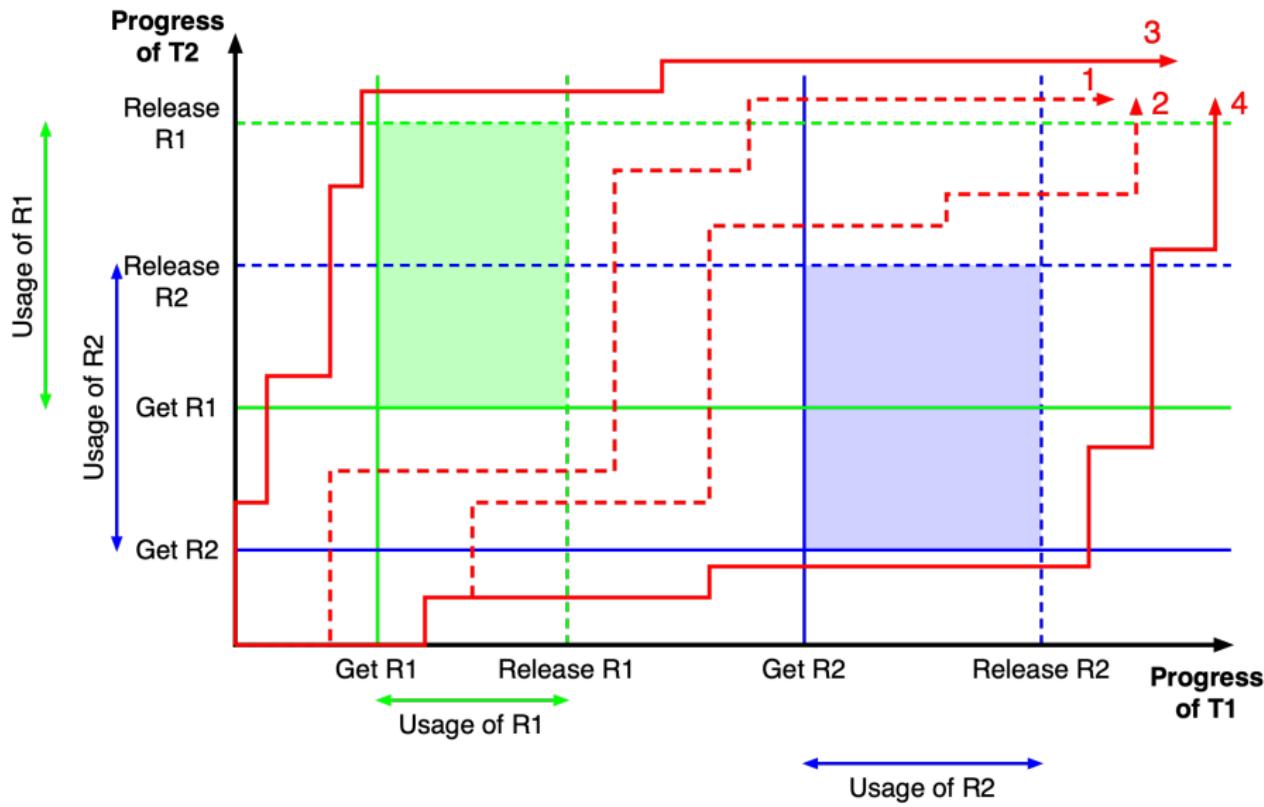
- Pokud však filosof musí alokovat vidličky ve vzrůstajícím pořadí
⇒ **řešení bez uváznutím**.



Příklad 1: Jak předejít vzniku uváznutí

- **Předpokládejme, že máme dvě vlákna T1 a T2.**
 - ▶ Vlákno T1 si alokuje dva prostředky v pořadí R1, R2.
 - ▶ Vlákno T2 si alokuje stejné prostředky v opačném pořadí R2, R1.
- **Průběh používání prostředků můžeme znázornit 2D grafem** (viz. následující grafy).
 - ▶ Vodorovná osa reprezentuje operace prováděné vláknem T1.
 - ▶ Svislá osa reprezentuje operace prováděné vláknem T2.
- Protože každý prostředek může být v jednom okamžiku používán pouze jedním vláknem \Rightarrow **vlákna musí obejít zelenou a modrou oblast.**
- **Trajektorie 1 až 4**
 - ▶ představují postupnou alokaci prostředků **bez uváznutí**.

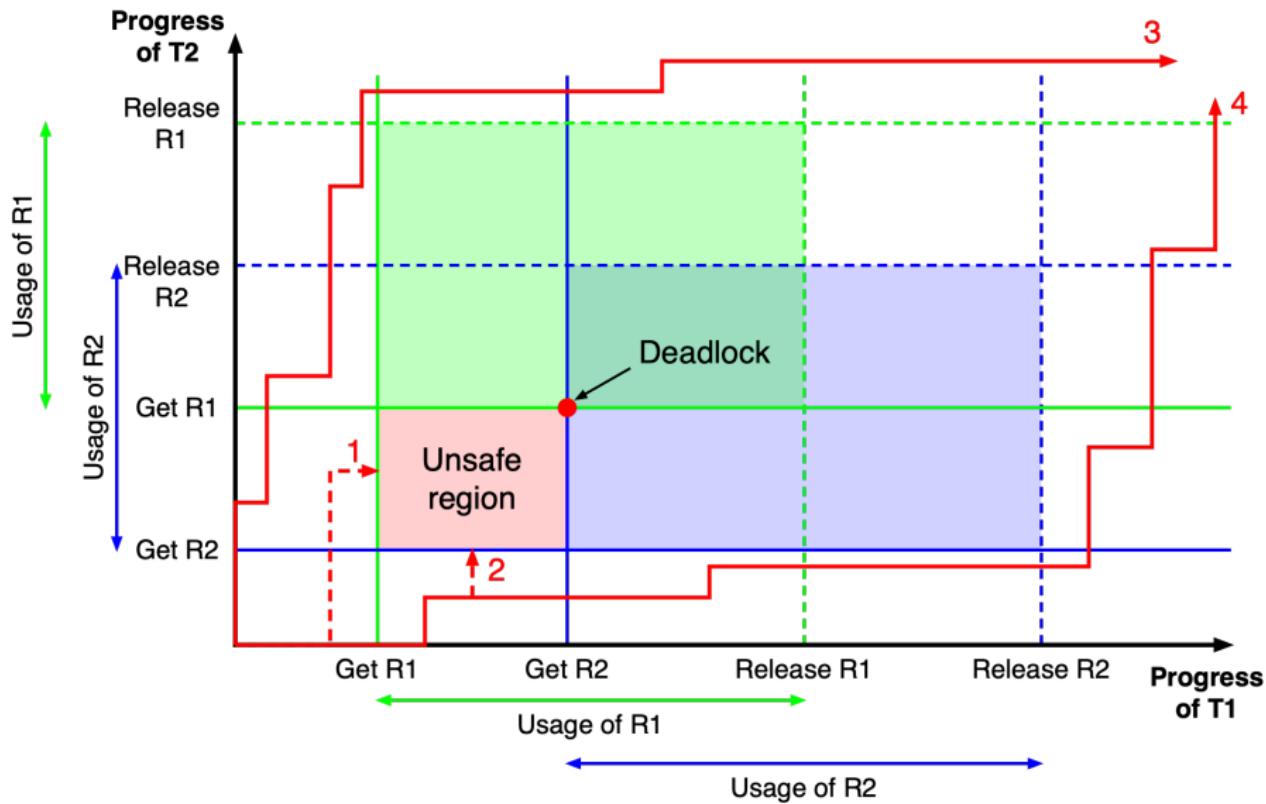
Příklad 1: Jak předejít vzniku uváznutí



Příklad 2: Jak předejít vzniku uváznutí

- Pokud se změní doba, po kterou vlákno používá prostředek
⇒ může dojít uváznutí.
- Trajektorie 1 a 2
 - ▶ Červená oblast (unsafe region) představuje pro vlákna past
⇒ pokud do ní vstoupí po určitém čase skončí uváznutím.
- Trajektorie 3 a 4
 - ▶ Pokud červenou, zelenou a modrou oblast vlákna obejdou
(vhodnou alokací prostředků) ⇒ uváznutí se vyhnou.

Příklad 2: Jak předejít vzniku uváznutí



Předcházení vzniku uváznutí

- **V příkladu 1 nenastal problém s uváznutím.**
 - ▶ Důvodem bylo to, že vlákno T1 uvolnilo prostředek R1 dříve než požádalo o prostředek R2 ⇒ nevznikla nebezpečná oblast (unsafe region).
 - ▶ Důležité je vědět, které prostředky budou vlákna používat a které z nich současně.
- **Jaké znalosti o požadavcích vláken potřebujeme znát?**
 - ▶ Předem neznáme požadavky vláken na prostředky
⇒ uváznutí nelze předejít.
 - ▶ Předem známe
 - ★ požadavky vláken na prostředky
⇒ uváznutí lze předejít, ale prostředky nebudou efektivně využity (v příkladu 1 by nebyly povoleny trajektorie 1 a 2).
 - ★ požadavky vláken na prostředky + společné používání prostředků
⇒ uváznutí lze předejít a využití prostředků bude lepší
(v příkladu 1 by byly povoleny všechny trajektorie 1 až 4).

Předcházení vzniku uváznutí

• Popis aktuálního rozdělení prostředků v systému

- ▶ V systému je n vláken a m různých typů prostředků.
- ▶ **Vektor existujících prostředků E** (existence vector)
- ▶ **Vektor volných prostředků F** (free vector)
- ▶ **Matice požadavků Q** (request matrix)
 - ★ Obsahuje informaci o celkových požadavcích vláken na prostředky.
- ▶ **Matice přidělených prostředků A** (allocation matrix)
 - ★ Obsahuje informaci o aktuálně alokovaných prostředcích.
- ▶ **Matice chybějících prostředků M** (missing matrix)
 - ★ Obsahuje informaci o prostředcích, které vláknům aktuálně chybí.

$$E = [E_1, \dots, E_m]$$

$$F = [F_1, \dots, F_m]$$

$$Q = \begin{bmatrix} Q_{11} & Q_{12} & \dots & Q_{1m} \\ Q_{21} & Q_{22} & \dots & Q_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ Q_{n1} & Q_{n2} & \dots & Q_{nm} \end{bmatrix} \quad A = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1m} \\ A_{21} & A_{22} & \dots & A_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \dots & A_{nm} \end{bmatrix}$$

$$M = Q - A$$

- **Pro rozložení prostředků v systému by mělo platit**

- ▶ Všechny prostředky jsou buď volné nebo přidělené vláknům

$$E_i = F_i + \sum_{k=1}^n A_{ki} \quad \text{pro všechna } i = 1, \dots, m.$$

- ▶ Žádné vlákno nepožaduje více prostředků než kolik jich je v systému

$$Q_{ki} \leq E_i \quad \text{pro všechna } k = 1, \dots, n \text{ a } i = 1, \dots, m.$$

- ▶ Žádné vlákno si nealokuje více prostředků než požadovalo

$$A_{ki} \leq Q_{ki} \quad \text{pro všechna } k = 1, \dots, n \text{ a } i = 1, \dots, m.$$

• Předpoklady

- ▶ Předem známe všechny požadavky vláken na prostředky, které budou během své existence potřebovat.
- ▶ Pokud jim tyto prostředky přidělíme \Rightarrow vlákna tyto prostředky po určité době uvolní.

• Bezpečný stav

- ▶ Stav, ve kterém existuje taková posloupnost přidělování prostředků vláknům, která garantuje postupné uspokojení potřeb všech vláken.

• Bankéřův algoritmus

- ▶ Když vlákno požádá o prostředek
 - ★ prostředek bude přidělen pokud systém zůstane v bezpečném stavu,
 - ★ jinak bude vlákno zablokováno a bude čekat na prostředek.

• Jak zjistit, že stav systému je bezpečný?

- ① Existuje vlákno, které lze uspokojit volnými prostředky?
 - ★ Vlákno existuje
⇒ vlákno postupně uvolní všechny prostředky a opakujeme bod 1.
 - ★ Vlákno nexistuje
⇒ pokračujeme na bod 2.
- ② Byla uspokojena všechna vlákna?
 - ★ Ano ⇒ bezpečný stav.
 - ★ Ne ⇒ není to bezpečný stav.

Příklad: Bankéřův algoritmus

- Je tento stav systému bezpečný?

	R1	R2	R3
T1	3	2	6
T2	6	1	3
T3	3	1	4
T4	4	2	2

Request matrix Q

	R1	R2	R3
T1	1	0	0
T2	6	1	2
T3	2	1	1
T4	0	0	2

Allocation matrix A

	R1	R2	R3
9	3	6	
Existence vector E			
	R1	R2	R3
0	1	1	
Free vector F			

- S volnými prostředky $F = [0, 1, 1]$ můžeme uspokojit vlákno T2.
 - Po určité době vlákno T2 skončí a uvolní své alokované prostředky.

	R1	R2	R3
T1	3	2	6
T2	6	1	3
T3	3	1	4
T4	4	2	2

Request matrix Q

	R1	R2	R3
T1	1	0	0
T2	6	1	2
T3	2	1	1
T4	0	0	2

Allocation matrix A

	R1	R2	R3
2	2	6	
T2	0	0	1
T3	1	0	3
T4	4	2	0
Missing matrix M			
	R1	R2	R3
9	3	6	
Existence vector E			
	R1	R2	R3
0	1	1	
Free vector F			

Příklad: Bankéřův algoritmus

- ② S volnými prostředky $F = [6, 2, 3]$ můžeme postupně uspokojit vlákna T3 a T4.
- ▶ Po určité době vlákna T3 a T4 skončí a uvolní své alokované prostředky.

	R1	R2	R3
T1	3	2	6
T2	0	0	0
T3	3	1	4
T4	4	2	2

Request matrix Q

	R1	R2	R3
T1	1	0	0
T2	0	0	0
T3	2	1	1
T4	0	0	2

Allocation matrix A

	R1	R2	R3
T1	2	2	6
T2	0	0	0
T3	1	0	3
T4	4	2	0

Missing matrix M

	R1	R2	R3
Free vector F	6	2	3

- ③ S volnými prostředky $F = [8, 3, 6]$ můžeme uspokojit i poslední vlákno T1. ⇒ systém je v bezpečném stavu.

	R1	R2	R3
T1	3	2	6
T2	0	0	0
T3	0	0	0
T4	0	0	0

Request matrix Q

	R1	R2	R3
T1	1	0	0
T2	0	0	0
T3	0	0	0
T4	0	0	0

Allocation matrix A

	R1	R2	R3
T1	2	2	6
T2	0	0	0
T3	0	0	0
T4	0	0	0

Missing matrix M

	R1	R2	R3
Free vector F	8	3	6

- Předchozí strategie (prevence/předcházení uváznutí) jsou konzervativní a jsou založené na omezování přístup vláken k prostředků.
- Pokud však nemáme informace o tom, které prostředky budou vlákna používat nebo jak je budou používat, pak tyto strategie nelze použít ⇒ k uváznutí může dojít.
- **Následující strategie s uváznutím počítá** a obsahuje dvě fáze.
 - ① Detekci uváznutí
 - ★ V systému jsou prováděné pravidelné kontroly, které se snaží odhalit existující uváznutí.
 - ② Zotavení z uváznutí
 - ★ Zotavení je založené na tom, že se část prostředků "uvolní" a tím se poruší čekání vláken ve smyčce ⇒ uváznutí se odstraní.
- Zatímco detekci lze implementovat poměrně jednoduše, samotné zotavení může být poměrně složité.

Detekce uváznutí

• Popis aktuálního stavu systému

- ▶ vektor existujících prostředků E (existence vector),
- ▶ vektor volných prostředků F (free vector),
- ▶ matici přidělených prostředků A (allocation matrix),
- ▶ matici požadavků Q^c (current request matrix),
 - ★ Obsahuje informaci o prostředcích aktuálně požadovaných jednotlivými vlákny.

• Algoritmus pro detekci uváznutí

- 1 Vytvoříme kopii C vektoru F (aktuálně volné prostředky).
- 2 Na začátku jsou všechna vlákna neoznačená.
- 3 Označíme všechna vlákna, která nechtějí žádný prostředek.
- 4 Existuje vlákno, které lze uspokojit prostředky z C ?
 - ★ **Vlákno existuje**
⇒ vlákno označíme, jeho alokované prostředky přičteme k vektoru C a opakujeme bod 4.
 - ★ **Vlákno neexistuje**
⇒ pokračujeme na bod 5.
- 5 Byla označena všechna vlákna?
 - ★ Ano ⇒ k uváznutí nedošlo.
 - ★ Ne ⇒ uváznutí nastalo (neoznačená vlákna jsou uvázlá).



Příklad: Detekce uváznutí

- Nachází se v systému uváznutí?

	R1	R2	R3	R4	R5
T1	1	0	1	1	0
T2	1	1	0	0	0
T3	0	0	0	1	0
T4	0	0	0	0	0

Allocation matrix A

T1	0	1	0	0	1
T2	0	0	1	0	1
T3	0	0	0	0	1
T4	1	0	1	0	1

Current request matrix Q^C

R1	2	1	1	2	1
R2	0	0	0	0	1
R3	0	0	0	0	0

Existence vector E

R1	0	0	0	0	1
R2	0	0	0	0	0
R3	0	0	0	0	0

Free vector C

- ① Volnými prostředky $C = [0, 0, 0, 0, 1]$ lze uspokojit vlákno T3.
- ▶ Vlákno T3 označíme a jeho prostředky přičteme k vektoru C .
 - ▶ S prostředky $C = [0, 0, 0, 1, 1]$ nelze uspokojit žádný další proces
⇒ vlákna T1, T2 a T4 jsou uvázlá.

	R1	R2	R3	R4	R5
T1	1	0	1	1	0
T2	1	1	0	0	0
T3	0	0	0	1	0
T4	0	0	0	0	0

Allocation matrix A

T1	0	1	0	0	1
T2	0	0	1	0	1
T3	0	0	0	0	1
T4	1	0	1	0	1

Current request matrix Q^C

R1	2	1	1	2	1
R2	0	0	0	0	1
R3	0	0	0	0	0

Existence vector E

R1	0	0	0	1	1
R2	0	0	0	0	0
R3	0	0	0	0	0

Free vector C

● Standardní nástroje OS pro detekci uváznutí

- ▶ Příkazy pro zobrazí stavu vláken (např. příkaz `ps`,...).
- ▶ Příkazy, které dokáží zobrazit zásobníky vláken (např. příkaz `gstack`, `debugger`,...).
- ▶ Různé aplikace pro ladění a profilování programů (např. `Valgrind`,...).

• **Ukončení všech uvázlých vláken**

- ▶ Typické řešení v OS pokud uživatel/administrátor zjistí problém (např. pomocí zaslání signálu).

• **Postupné ukončování uvázlých vláken**

- ▶ Postupně ukončujeme uvázlá vlákna dokud je v systému detekované uváznutí.

• **Zotavení pomocí návratu restartu**

- ▶ Princip této strategie je založen na znovu spuštění uvázlých vláken z některého z předchozích stavů.
- ▶ Toto řešení vyžaduje, aby byl v systému implementovaný mechanismus návratu (rollback) a opětovného spuštění (restart).
- ▶ Systém si pravidelně ukládá svůj stav (důležité informace o systému) tak, aby později bylo možné obnovit systém do některého z předchozích stavů.
- ▶ Díky nedeterminaci paralelního zpracování, je pravděpodobnost výskytu stejného uváznutí relativně malá.

Použité zdroje

- ① A. S. Tanenbaum, H. Bos: *Modern Operating Systems (4th edition)*, Pearson, 2014.
- ② W. Stallings: *Operating Systems: Internals and Design Principles (9th edition)*, Pearson, 2017.
- ③ A. Silberschatz, P. B. Galvin, G. Gagne: *Operating System Concepts (9th edition)*, Wiley, 2012.

Operační systémy

Implementace procesů a vláken. Plánování vláken.

Jan Trdlička



České vysoké učení technické v Praze, Fakulta informačních technologií
Katedra počítačových systémů

<https://courses.fit.cvut.cz/BI-OSY>

Obsah přednášky

1 Implementace procesů/vláken

- Process control block
- Thread control block
- Implementace vláken

2 Plánování vláken

- Typy aplikací
- Cíle plánování
- Typy vláken
- Kdy plánujeme?

3 Strategie plánování

- Plánování s odnímáním
 - Round-robin (RR)
 - Prioritní RR se statickou prioritou
 - Prioritní RR s dynamickou prioritou
- Kooperativní plánování
 - First-come-first-served (FCFS)

• Tabulka procesů

- ▶ Jádro OS si udržuje informace o procesech nejčastěji pomocí nějaké varianty zřetězeného seznamu struktur, který můžeme nazývat "tabulkou procesů".

• Process control block (PCB)

- ▶ Jedna položka tabulky (struktura) reprezentuje všechny nezbytné informace, které si jádro OS musí pamatovat pro jeden proces, a historicky se nazývá process control block.

• Kolik procesů/vláken může být maximálně vytvořeno?

▶ V rámci celého systému

- ★ Většina OS odvozuje maximální počet procesů/vláken např. od velikosti fyzické paměti (např. Linux, Solaris,...) a může být definován např. pomocí parametrů jádra.

▶ Jedním uživatelem

- ★ Z důvodu bezpečnosti je většinou maximální počet procesů/vláken pro běžné uživatele nastaven (např. ochrana před fork-bombou).
- ★ V unixových systémech např. pomocí příkazů ulimit, sysctl, rctladm,...

Příklad: Maximální počet procesů/vláken v Linuxu

- Kolik vláken lze maximálně vytvořit v systému?

```
u1@linux:~> cat /proc/sys/kernel/threads-max  
15523
```

- Kolik vláken může běžný uživatel u1 vytvořit?

```
u1@linux:~> ulimit -u  
7761
```

- Kolik vláken má běžný uživatel právě vytvořeno?

```
u1@linux:~> ps -Lu u1 | tail -n+2 | wc -l  
13
```

- Nastavíme nový limit.

```
u1@linux:~> ulimit -u 100  
u1@linux:~> ulimit -u  
100
```

- ▶ Ověříme jeho funkčnost pomocí bash fork-bomby.

```
u1@linux:~> f(){ f | f & } ; f  
-bash: fork: retry: Resource temporarily unavailable  
...
```

- ▶ Ověříme, že limit zafungoval, a ukončíme fork-bombu.

```
root@linux:~> ps -Lu u1 | tail -n+2 | wc -l  
100  
root@linux:~> pkill -u u1
```

Process control block

- Obsahuje všechny nezbytné informace, které si OS musí pamatovat o procesu.
- **Informace pro identifikaci procesu**
 - ▶ číslo procesu (PID), číslo rodičovského proces (PPID),
 - ▶ číslo seance (SID), číslo úlohy, číslo projektu,
 - ▶ jméno procesu,...
- **Informace související se identitou procesu/bezpečností**
 - ▶ vlastník procesu (EUID, RUID,...),
 - ▶ příslušnost ke skupinám (GID),
 - ▶ privilegia přiřazená procesu, access token,...
- **Informace o alokovaných prostředcích**
 - ▶ paměť
 - ★ informace o přidělené fyzické paměti,
 - ★ informace nutné pro překlad logických adres na fyzické,...
 - ▶ otevřené soubory (tabulka deskriptorů souborů)
 - ▶ prostředky pro mezi procesovou komunikaci (IPC)
 - ★ synchronizační nástroje (semafory, roury, signály,...),...

- Pro každé vlákno daného procesu si OS musí pamatovat řadu informací, které jsou uloženy v různých datových strukturách. Tyto struktury budeme označovat jako **thread control block** a obsahují následující informace.
 - ▶ **informace pro identifikaci vlákna**
 - ★ číslo vlákna (TID).
 - ▶ **informace pro přepínání kontextu**
 - ★ hodnoty viditelných registrů CPU,
 - ★ hodnoty řídících a stavových registrů CPU (čítač instrukcí, ...),
 - ★ ukazatel na zásobník, ...
 - ▶ **informace pro plánování vláken**
 - ★ typ plánovacího algoritmu,
 - ★ priorita,
 - ★ stav vlákna,
 - ★ informace o událostech, na které vlákno čeká,
 - ★ využitý čas CPU,
 - ★ důvod posledního přepnutí kontextu, ...

Process control block

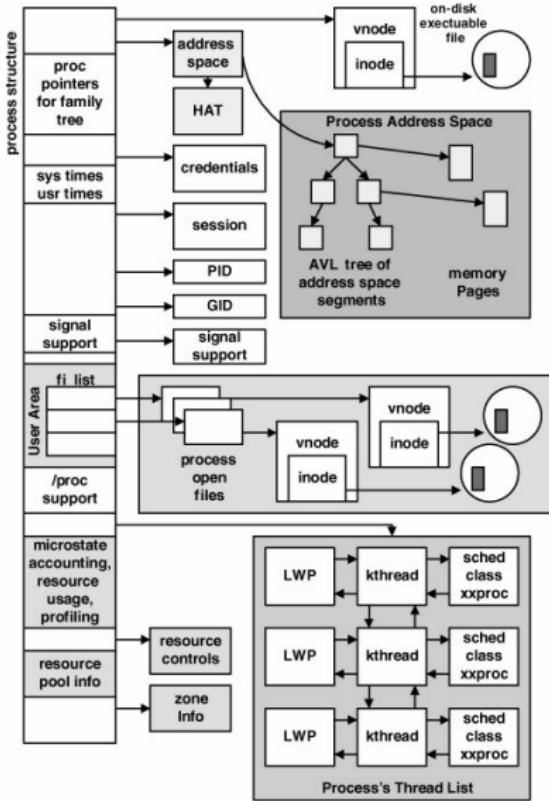
• Jak zjistit strukturu PCB na konkrétním OS?

- ▶ dokumentace k danému OS (MS Windows, Linux, Solaris),...
- ▶ publikace typu OS internals
 - ★ Windows Internals Part 1, Part 2 (7th Edition),
 - ★ Linux Kernel Development (3rd Edition),
 - ★ MacOS and iOS Internals,
 - ★ Solaris Internals: Solaris 10 ... (2nd Edition),...
- ▶ zdrojový kód, pokud je k dispozici (např. Linux kernel),
- ▶ hlavičkové soubory
 - ★ MS Windows: `winternl.h`,...
 - ★ Linux: `sched.h`,...
 - ★ Solaris: `proc.h`, `thread.h`,...

• Jak zjistit informace z PCB pro určitý process?

- ▶ aplikace: Task Manager, Process Explorer (MS Windows),
- ▶ příkazy: `ps`, `top`, `lsof`, `gstack`,...
- ▶ OS API: `getpid`, `gettid`, `getuid`,...
- ▶ debugger: WinDbg (MS Windows), `gdb` (Linux), `mdb` (Solaris),...
- ▶ adresář `/proc`: v OS unixového typu,...

Příklad: Struktura procesu proc_t v Solarisu



```
user@solaris:~> cat /usr/include/sys/proc.h
/*
 * One structure allocated per active process. It contains all
 * data needed about the process while the process may be swapped
 * out. Other per-process data (user.h) is also inside the proc structure.
 * Lightweight-process data (lwp.h) and the kernel stack may be swapped out.
 */
typedef struct proc {
    /*
     * p_exec is protected by p_lock
     */
    struct vnode *p_exec;           /* pointer to a.out vnode */

    /*
     * Fields requiring no explicit locking
     */
    struct as *p_as;               /* process address space pointer */
    struct flock *p_lockp;          /* ptr to proc struct's mutex lock */
    kmutex_t p_crllock;            /* lock for p_cred */
    struct cred *p_cred;           /* process credentials */

    /*
     * Fields protected by pidlock
     */
    int     p_swapcnt;             /* number of swapped out lwp */
    char    p_stat;                /* status of process */
    char    p_wcode;                /* current wait code */
    ushort_t p_pidflag;            /* flags protected only by pidlock */
    int     p_wdata;                /* current wait return value */
    pid_t   p_ppid;                /* process id of parent */
    struct proc *p_link;           /* forward link */
    struct proc *p_parent;          /* ptr to parent process */
    struct proc *p_child;           /* ptr to first child process */

    ...
    /*
     * Per process signal stuff.
     */
    k_sigset_t p_sig;              /* signals pending to this process */
    k_sigset_t p_extsig;            /* signals sent from another ... */
    k_sigset_t p_ignore;            /* ignore when generated */
    k_sigset_t p_siginfo;           /* gets signal info with signal */
    struct sigqueue *p_sigqueue;    /* queued siginfo structures */
    struct sigqhdr *p_sigqhdr;      /* hdr to sigqueue structure pool */
    struct sigqhdr *p_sighdr;        /* hdr to signify structure pool */
    uchar_t p_stopsig;              /* jobcontrol stop signal */
    ...
} proc_t;
```

Příklad: Debuggování jádra Solarisu

- Zjistění informace o procesech pomocí debagování jádra.

```
root@solaris:~> echo "::ps" | mdb -k
S     PID      PPID      PGID      SID      UID      FLAGS          ADDR NAME
R       0         0         0         0      0x00000001 ffffffc063270 sched
R       1         0         0         0      0x4a004000 fffffa1c002c571e8 init
.
.
R   1436         1     1302     1302    100 0x4a004000 fffffa1c00d2c0238 gnome-terminal
R   1439     1436     1439     1439    100 0x4a014000 fffffa1c00c4dc2b0 bash
R  26200     1439    26200     1439      0 0x5a006000 fffffa1000def87c8 su
R 26201    26200    26201     1439      0 0x4a014000 fffffa1c0089ca088 bash
.
.
```

- Zobrazení obsahu PCB konkrétního procesu s PID=26201.

```
root@solaris:~> echo "fffffa1c0089ca088 ::print -ta proc_t" | mdb -k
fffffa1c0089ca088 proc_t {
    fffffa1c0089ca088 struct vnode *p_exec = 0xfffffa1c00c586200
    fffffa1c0089ca090 struct as *p_as = 0xfffffa1000df1a8d8
    fffffa1c0089ca098 struct plock *p_lockp = 0xfffffa1c001d59680
    fffffa1c0089ca0a0 kmutex_t p_crlock = {
        fffffa1c0089ca0a0 void *[1] _opaque = [ 0 ]
    }
    fffffa1c0089ca0a8 struct cred *p_cred = 0xfffffa1c0049c5eb0
    fffffa1c0089ca0b0 int p_swapcnt = 0
    fffffa1c0089ca0b4 char p_stat = '\002'
    fffffa1c0089ca0b5 char p_wcode = '\0'

.
.
fffffa1c0089ca0c8 struct proc *p_parent = 0xfffffa1000def87c8
fffffa1c0089ca0d0 struct proc *p_child = 0xfffffa1000df50a18
.
.
```



Příklad: Obsah adresáře /proc v Linuxu

- Do adresář /proc je připojen pseudo systém souborů (jeho obsah existuje pouze v paměti).

```
ul1@linux:~> mount | grep /proc
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
```

- Prostřednictvím tohoto systému souborů jádro OS umožňuje získat/nastavit hodnoty některých parametrů jádra.
 - Parametr threads-max definuje maximální počet vláken, která lze v systému vytvořit.

```
ul1@linux:~> cat /proc/sys/kernel/threads-max
31256
```

- V podadresářích, které mají jméno stejné jako je PID procesu, jsou uložené informace o konkrétních procesech.
 - Jak zjistit limity nastavené pro naš aktuální shell?

```
ul1@linux:~> ps
  PID TTY          TIME CMD
 36282 pts/0    00:00:00 bash
ul1@linux:~> cat /proc/36282/limits
Limit                      Soft Limit                  Hard Limit          Units
. . .
Max stack size              8388608                  unlimited           bytes
Max processes                100                      100                 processes
. . .
```



- Historicky rozlišujeme dva způsoby implementace vláken

- ▶ v uživatelském prostoru (user-level threads),
 - ▶ v jádru OS (kernel-level threads).

- Vlákna implementovaná v uživatelském prostoru

- ▶ Historicky používaná v OS, které podporovaly pouze proces s jedním "vláknem".
- ▶ Vlákna jsou kompletně **podporována/spravována v uživatelském prostoru pomocí "run-time" systému** (user-level knihovna).
- ▶ Uživatelská vlákna používají **kooperativní plánování** (po určitém čase vlákno předá řízení zpět run-time systému pomocí příslušné funkce).
- ▶ Jádro OS nemá "žádnou" informaci o uživatelských vláknech v jednotlivých procesech a **spravuje je jako proces s jedním vláknem**.

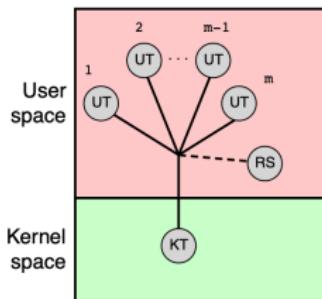
- ▶ **Vlastnosti**

- ▶
 - ★ Jednoduchá/rychlá správa: vytvoření, přepínání, synchronizace vláken je v uživatelském prostoru bez zásahu jádra OS.
 - ★ Vlákna jednoho procesu jsou **mapována na jedno jádro CPU**.
 - ★ Blokující volání v jednom vláknu **zablokuje i ostatní vlákna procesu**.
- ▶ V praxi je nutná aspoň minimální podpora jádra OS tak, aby blokující volání nezablokovala všechna vlákna v procesu.

Implementace vláken

• Vlákna implementovaná v uživatelském prostoru

- Tato implementace je někdy označovaná jako model many-to-one, ve kterém je více uživatelských vláken namapováno na jedno kernel vlákno.



(UT) User-level thread

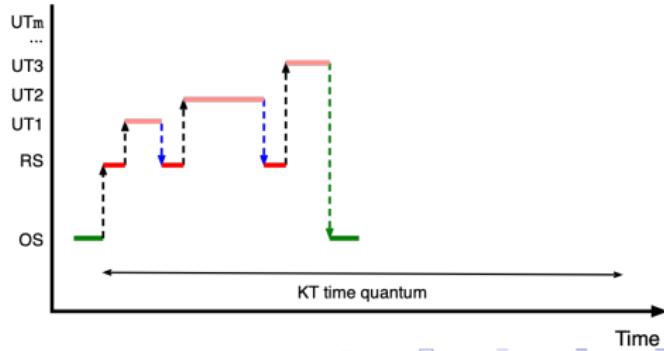
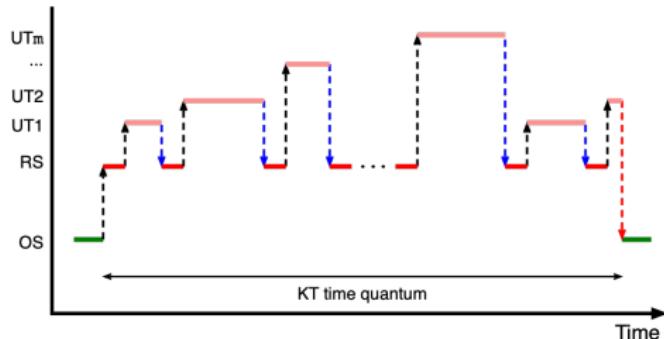
(RS) Run-time system

(KT) Kernel-level thread

— User thread yields CPU core

— Interrupt from timer

— User thread uses blocking system call



• Vlákna implementovaná v jádře OS

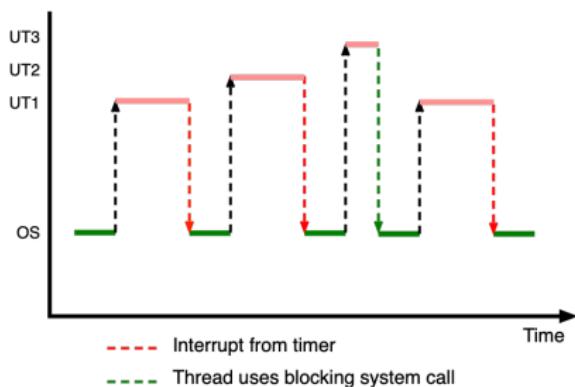
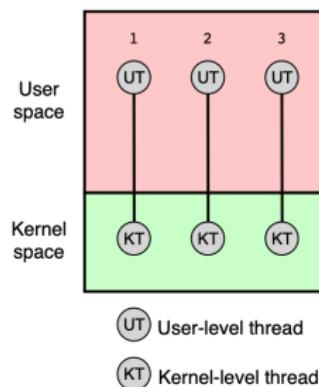
- ▶ Typická implementace v současných OS s podporou vláken (MS Windows, Linux, Solaris, MacOS,...).
- ▶ Vlákna jsou podporována/spravována přímo jádrem OS.
- ▶ Jádro OS
 - ★ spravuje jeden PCB pro každý proces,
 - ★ spravuje jeden TCB pro každé vlákno,
 - ★ poskytuje systémová volání pro vytváření/správu vláken z uživatelského prostoru.

▶ Vlastnosti

- ★ Jádro má všechny informace o všech vláknech
⇒ jádro OS přiděluje jednotlivá jádra CPU jednotlivým vláknům na určitou dobu.
- ★ Pokud zavolá vlákno **blokující** systémové volání ⇒ zablokuje se **pouze toto vlákno**.
- ★ Vytvoření/ukončení vlákna, systémové volání, přepnutí kontextu,...
⇒ představuje **přepnutí z user modu do kernel modu**.

• Vlákna implementovaná v jádře OS

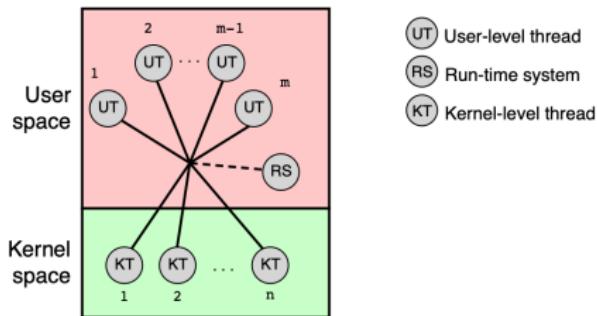
- Tato implementace je někdy označovaná jako **model one-to-one**, ve kterém je jedno uživatelské vlákno namapováno na jedno kernel vlákno.



Implementace vláken

• Hybridní implementace

- ▶ Tato implementace je někdy označovaná jako **model many-to-many**, ve kterém je m uživatelských vláken namapováno na n kernel vláken .
- ▶ Jedná se o kombinaci předchozích implementací.



• Nejednoznačná terminologie

- ▶ V současných OS jsou vlákna implementována v jádru OS.
- ▶ Občas jsou v různých dokumentacích zmiňována "uživatelská vlákna", která jsou implementována pomocí knihovny. Většinou je tím ale myšlena knihovna, která implementuje model vláken "one-to-one" (v jádru OS).
- ▶ Samotné jádro OS je implementováno jako vícevláknové a někdy je pod pojmem "kernel vlákno" myšleno vlákno, které implementuje příslušnou funkci jádra.

Příklad: Implementace procesů/vláken v Solarisu

• Vlákno

- ▶ User-level thread (ULT): implementovaný pomocí knihovny vláken v adresovém prostoru procesu (neviditelný pro OS).
- ▶ Kernel threads: základní jednotky, které jsou plánovány a spuštěny na jednotlivých jádrech CPU.
- ▶ Lightweight process (LWP): mapování mezi ULT a kernel vlákny. Od Solarisu 8 (rok 2000) už je podporován pouze model one-to-one.

• Proces

- ▶ Normální Unixový proces.

• Úloha (task)

- ▶ Množina procesů jednoho uživatele, pro které lze nastavit limity (maximální počet vláken, procesů,...).

• Projekt (project)

- ▶ Množina procesů jednoho/více uživatelů, pro které lze nastavit limity.

• Zóna (zone)

- ▶ Virtuální instance OS (množina systémových a uživatelských procesů, které využívají pouze zdroje přidělené dané zóně).

• Limity, které lze nastavit viz. příkaz `rctladm`.



- **Vlákno**

- ▶ Fiber: vlákno spravované celé v uživatelském prostoru.
- ▶ Thread: jednotka plánována jádrem.

- **Proces**

- ▶ Jednotka, která alokuje zdroje. Každý proces má aspoň jedno vlákno (thread).

- **Job**

- ▶ Množina procesů, pro které lze nastavit limity v rámci OS (maximální počet procesů, čas CPU, paměť, ...).

- V okamžiku, kdy se uvolní jádro CPU, musí OS vybrat "vhodné" vlákno ve stavu "Ready", a umožní mu používat toto jádro po určitou dobu.
- V OS je obvykle implementováno několik různých plánovacích strategií a uživatel/administrátor může pro různé aplikace nastavit vhodnou strategii, popřípadě určí, na kterých jádrech CPU poběží.
- **Typy aplikací z hlediska plánování vláken**
 - ▶ Dávkové úlohy (aplikace běžící na pozadí)
 - ★ obvykle zpracovávají data, která načítají ze/zapisují do souborů, a nekomunikují přímo s uživatelem,
 - ★ vyžadují relativně hodně CPU výkonu.
 - ▶ Interaktivní aplikace
 - ★ reagují na události, které přicházejí např. z GUI, CLI, síťového rozhraní,...
 - ▶ Úlohy reálného času
 - ★ většinou nevyžadují mnoho CPU výkonu, ale je nutné zajistit/garantovat co nejrychlejší reakci na událost (např. aplikace řídící technologický proces).

• Z hlediska systému

- ▶ **Spravedlivost:** každý uživatel/proces/vlákno získá úměrnou část CPU výkonu.
- ▶ **Vyváženost:** rozložit zátěž rovnoměrně na všechny části systému.
- ▶ **Prosazení strategie:** možnost uplatnit zvolenou plánovací strategii.

• Dávkové úlohy

- ▶ **Doba zpracování (turnaround time):** čas, který uplyne od spuštění do ukončení úlohy.

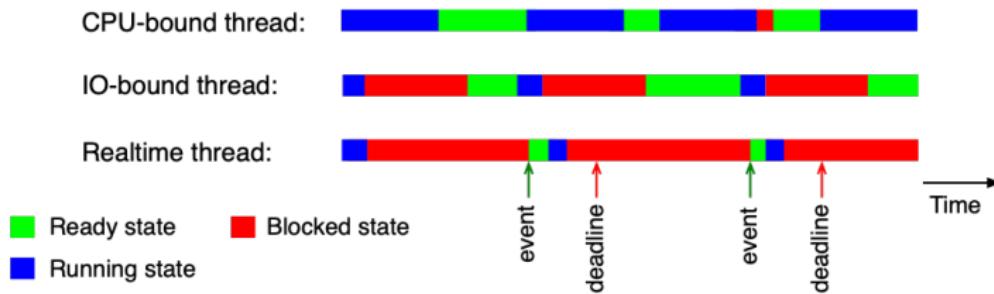
• Interaktivní aplikace

- ▶ **Doba odezvy (response time):** čas, který uplyne od okamžiku zadání požadavku (události) do okamžiku první reakce.
- ▶ **Přiměřenost (proportionality):** splnit očekávání uživatele.

• Úlohy reálného času

- ▶ **Garantování odezvy (meeting deadlines):** zabránění ztrátě dat.
- ▶ **Předvídatelnost:** zamezení degradaci výkonu.

Typy vláken



- Z hlediska plánování OS většinou rozlišuje tři typy vláken.
 - ➊ **Vlákna orientovaná na CPU (CPU-bound threads)**
 - ★ CPU využíváno po dlouhou dobu, málo blokujících operací.
 - ➋ **Vlákna orientovaná na V/V (I/O-bound threads)**
 - ★ CPU využito po krátkou dobu, hodně blokujících operací.
 - ➌ **Vlákna reálného času (Realtime threads)**
 - ★ Vlákno musí zareagovat na událost během daného intervalu.
- První dva typy vyplývají z **předchozího chování vlákna** a během existence vlákna se může typ měnit.
- Třetí typ vyplývá ze **způsobu použití vlákna**, kdy je nutné zajistit rychlou reakci na příslušnou událost.

Kdy plánujeme?

- **V okamžiku, kdy běžící vlákno "odevzdá" jádro CPU.**
 - ▶ Vlákno končí svůj výpočet
 - ★ Vlákno dokončilo svojí "práci" a zavolalo příslušné systémové volání (např. `exit()`, ...)
 - ▶ Vlákno dobrovolně odevzdává jádro CPU
 - ★ Vlákno se dobrovolně vzdalo jádra CPU pomocí příslušného systémového volání (např. `pthread_yield()`, ...)
 - ▶ Vlákno volá blokující systémové volání/knihovní funkci
 - ★ např. `read()`, `write()`, `pthread_mutex()`, ...)
- **V okamžiku, kdy nastane nějaká událost**
 - ▶ Přerušení od
 - ★ V/V zařízení: dokončila se V/V operace a je nutné na to zareagovat,
 - ★ časovače: uplynulo časové kvantum přidělené danému vláknu, ...

Strategie plánování

● Plánování s odnímáním (preemptive scheduling)

- ▶ Jádro OS přidělí vláknu jádro CPU pouze na určitou dobu (časové kvantum) a po uplynutí této doby mu ho odebere.
- ▶ Vlastnosti
 - ★ Strategie vhodná pro vlákna v interaktivních systémech.
 - ★ Umožňuje "rozumným" způsobem sdílet CPU výkon systému.
 - ★ Dochází zde k častějšímu přepínání kontextu
⇒ horší využití CPU/lepší doba odezvy.

● Kooperativní plánování (cooperative scheduling)

- ▶ Jádro OS přidělí vláknu jádro CPU a vlákno ho používá, dokud ho samo neuvolní (např. dokončení výpočtu, blokující funkce,...).
- ▶ Vlastnosti
 - ★ Vlákno může blokovat systém a musí "spolupracovat" pouze když žádá službu jádra ⇒ tato strategie je vhodná pouze pro "prověřená/kernel" vlákna a není vhodná pro běžná uživatelská vlákna.
 - ★ Minimalizuje se počet přepnutí kontextu
⇒ lepší využití CPU/horší doba odezvy.

- Round-robin plánování (RR)



- ▶ Plánování s odnímáním, ve kterém vlákna ve stavu "Ready" čekají ve frontě FIFO, až jim bude přiděleno jádro CPU.
- ▶ Všem vláknům je jádro CPU propůjčováno na stejně velkou dobu (časové kvantum). Po uplynutí této doby je jádro CPU vláknům odebráno, změní se jejich stav na "Ready" a vlákna se zařadí na konec fronty (Ready queue).
- ▶ Volitelným parametrem plánování je velikost časového kvanta:
 - ★ Krátké čas. kvantum \Rightarrow horší využití CPU/krátká doba odezvy.
 - ★ Dlouhé čas. kvantum \Rightarrow dlouhá doba odezvy/lepší využití CPU.
 - ★ Rozumný kompromis je 10-50ms.

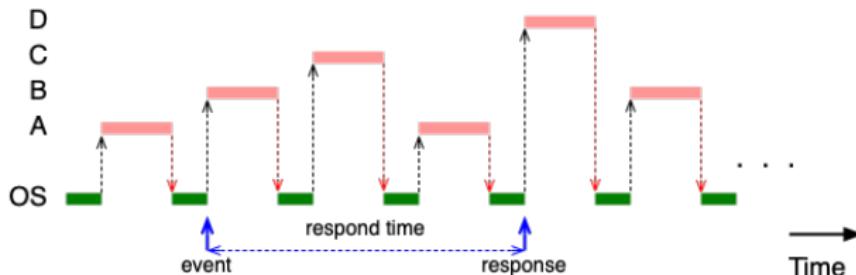
Příklad: Plánování s odnímáním RR

- **Předpokládejme následující parametry systému.**
 - ▶ Jádro CPU sdílejí pouze čtyři vlákna A,...,D pomocí strategie RR .
 - ★ Vlákna A, B a C jsou pouze ve stavech Ready a Running.
 - ★ Vlákno D je zablokováno (např. na mutexu).
 - ★ Těsně po přepnutí kontextu je vlákno D odblokováno.
 - ▶ Doba přepnutí kontextu je t_{cs} a časové kvantum je t_q .
- **Jaká je efektivita využití CPU a doba odezvy vlákna D na probuzení?**
 - 1 Pokud $t_{cs} = t_q = 10\text{ms}$.
 - 2 Pokud $t_{cs} = 10\text{ms}$ a $t_q = 90\text{ms}$.

Příklad: Plánování s odnímáním RR

- Předpokládejme následující parametry systému.
 - ▶ Jádro CPU sdílejí pouze čtyři vlákna A,...,D pomocí strategie RR .
 - ★ Vlákna A, B a C jsou pouze ve stavech Ready a Running.
 - ★ Vlákno D je zablokováno (např. na mutexu).
 - ★ Těsně po přepnutí kontextu je vlákno D odblokováno.
 - ▶ Doba přepnutí kontextu je t_{cs} a časové kvantum je t_q .
 - Jaká je efektivita využití CPU a doba odezvy vlákna D na probuzení?
- 1 Pokud $t_{cs} = t_q = 10\text{ms}$.
 - 2 Pokud $t_{cs} = 10\text{ms}$ a $t_q = 90\text{ms}$.

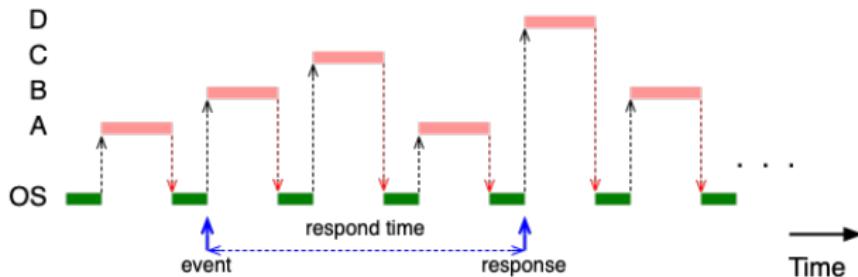
Core 1:



Příklad: Plánování s odnímáním RR

- Předpokládejme následující parametry systému.
 - ▶ Jádro CPU sdílejí pouze čtyři vlákna A,...,D pomocí strategie RR .
 - ★ Vlákna A, B a C jsou pouze ve stavech Ready a Running.
 - ★ Vlákno D je zablokováno (např. na mutexu).
 - ★ Těsně po přepnutí kontextu je vlákno D odblokováno.
 - ▶ Doba přepnutí kontextu je t_{cs} a časové kvantum je t_q .
 - Jaká je efektivita využití CPU a doba odezvy vlákna D na probuzení?
- 1 Pokud $t_{cs} = t_q = 10\text{ms}$.
 - 2 Pokud $t_{cs} = 10\text{ms}$ a $t_q = 90\text{ms}$.

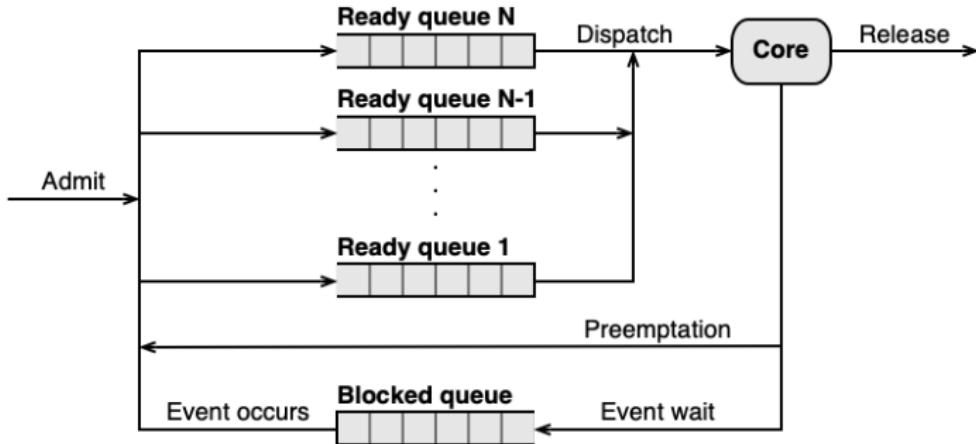
Core 1:



- 1 Efektivita: $t_q/(t_{cs} + t_q) \times 100 = 50\%$, odezva: $3 \times (t_{cs} + t_q) = 60\text{ms}$.
- 2 Efektivita: $t_q/(t_{cs} + t_q) \times 100 = 90\%$, odezva: $3 \times (t_{cs} + t_q) = 300\text{ms}$.

Plánování s odnímáním

- Prioritní RR plánování se statickou prioritou



- ▶ Plánování s odnímáním, ve kterém je každému vláknu přiřazena statická priorita (číslo $1, \dots, N$), která se během existence vlákna nemění (obvykle vyšší číslo reprezentuje vyšší prioritu).
- ▶ Volné jádro CPU je vždy přiřazeno vláknu s nejvyšší prioritou, které je na začátku příslušné fronty (Ready queue).
- ▶ "Ready" fronty jsou implementovány jako RR \Rightarrow FIFO.
- ▶ Časové kvantum může být pro různé priority různě velké.

Příklad: Prioritní RR plánování se statickou prioritou

- **Předpokládejme následující parametry systému.**

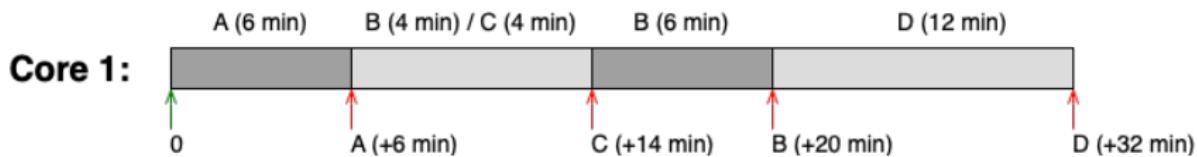
- ▶ Vlákna mají přidělenou statickou prioritu $1, \dots, 99$.
- ▶ Pro všechny priority je **velikost časového kvanta 100ms**.
- ▶ Režie na přepínání kontextu je zanedbatelná vzhledem k velikosti časového kvanta.
- ▶ Na jednom jádře CPU běží pouze **vlákna A, B, C a D**, která jsou spuštěna ve stejném okamžiku a jsou **orientovaná na CPU**.

Vlákno	Priorita	Požadovaný čas na jednom jádře [min]
A	99	6
B	75	10
C	75	4
D	50	12

- ① Za jak dlouho od spuštění jednotlivá vlákna skončí, pokud pouze ona poběží na jednom jádře CPU?
- ② Za jak dlouho od spuštění jednotlivá vlákna skončí, pokud pouze ona poběží na dvou jádrech CPU?

Příklad: Prioritní RR plánování se statickou prioritou

- 1 Za jak dlouho od spuštění jednotlivá vlákna skončí, pokud pouze ona poběží na jednom jádře CPU?

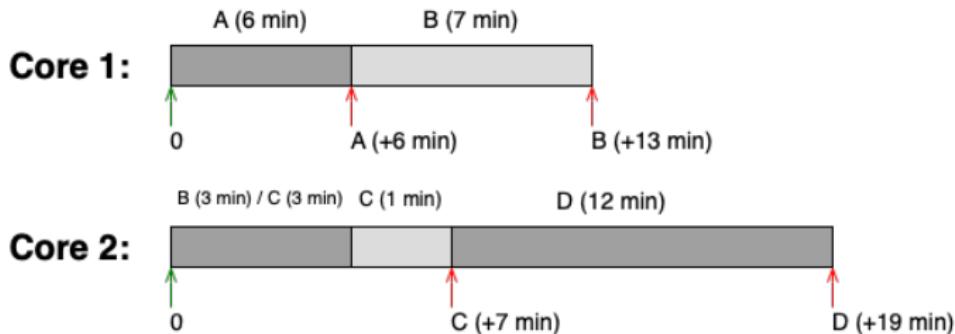


- Statická priorita

- ▶ Vlákňům s nízkou prioritou může být přiděleno jádro CPU až za "dlouhou dobu" ⇒ problémy s hladověním a pomalou odevzdu.

Příklad: Prioritní RR plánování se statickou prioritou

- ② Za jak dlouho od spuštění jednotlivá vlákna skončí, pokud pouze ona poběží na dvou jádrech CPU?

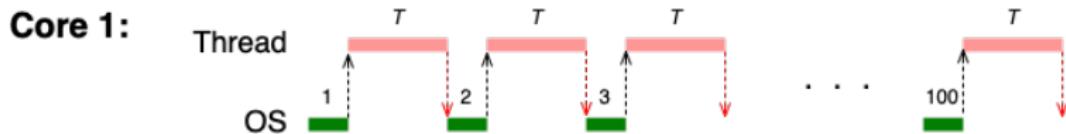


Příklad: Prioritní RR plánování se statickou prioritou

- Předpokládejme následující parametry systému.
 - ▶ Pro všechny priority je velikost časového kvanta T ms.
 - ▶ Na jednom jádru CPU běží pouze jedno vlákno orientované na CPU, které vyžaduje $100xT$ času jednoho jádra pro dokončení výpočtu.
- Ke kolika přepnutím kontextu dojde než se vlákno ukončí?

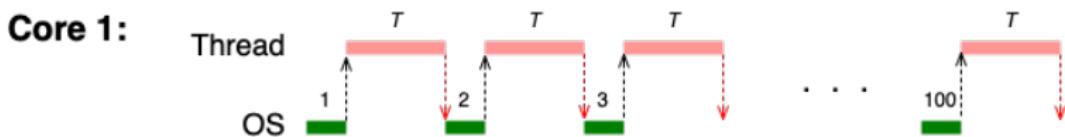
Příklad: Prioritní RR plánování se statickou prioritou

- Předpokládejme následující parametry systému.
 - ▶ Pro všechny priority je velikost časového kvanta T ms.
 - ▶ Na jednom jádru CPU běží pouze jedno vlákno orientované na CPU, které vyžaduje $100 \times T$ času jednoho jádra pro dokončení výpočtu.
- Ke kolika přepnutím kontextu dojde než se vlákno ukončí?



Příklad: Prioritní RR plánování se statickou prioritou

- Předpokládejme následující parametry systému.
 - ▶ Pro všechny priority je velikost časového kvanta T ms.
 - ▶ Na jednom jádru CPU běží pouze jedno vlákno orientované na CPU, které vyžaduje $100 \times T$ času jednoho jádra pro dokončení výpočtu.
- Ke kolika přepnutím kontextu dojde než se vlákno ukončí?



- Kontext se přepne v důsledku zpracování vlákna 100x.
- Fixní časové kvantum
 - ▶ Při plánování s fixním časovým kvantem se nezohledňuje chování vláken \Rightarrow vlákna orientovaná na CPU mohou vygenerovat velký počet přepnutí kontextu během své existence.

Prioritní RR s dynamickou prioritou

- Plánování s odnímáním, ve kterém se **priorita i velikost časového kvanta během existence vlákna může měnit**, tak aby bylo dosaženou určitého cíle.
- Nejčastějším cílem je **minimalizovat problém hladovění vláken, zlepšit odezvu a snížit počet přepnutí kontextu**.
- Strategie pro dosažení tohoto cíle může být následující
 - ▶ **Priorita se zvýší a časové kvantum sníží**
 - ★ pokud vlákno v posledním běhu nevyužilo celé své časové kvantum (vlákno orientované na V/V),
 - ★ pokud vlákno dlouho čeká na CPU (hrozí problém hladovění vlákna).
 - ▶ **Priorita se sníží a časové kvantum se zvýší**
 - ★ pokud vlákno v posledním běhu využilo celé své časové kvantum (vlákno orientované na CPU).
- Tato strategie je výchozí plánovací strategií v běžných OS.
- Její implementace je závislá na konkrétním OS.

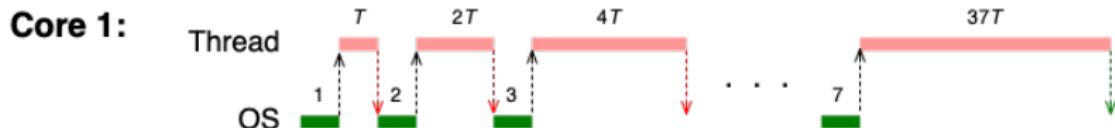
Příklad 3: Prioritní RR plánování

- **Předpokládejme následující parametry systému.**
 - ▶ Používá se dynamická prioritá s proměnným časovým kvantem.
 - ★ Při spuštění mají vlákna nastavenou prioritu P_0 a časové kvantum T .
 - ★ Pokud vlákno s aktuální prioritou P_i **využije celé časové kvantum**
⇒ v následujícím běhu poběží s prioritou $P_i - 1$ a dvojnásobným časovým kvantem.
 - ★ Pokud vlákno s aktuální prioritou P_i **nevyužije celé časové kvantum**
⇒ v následujícím běhu poběží s prioritou $P_i + 1$ a polovičním časovým kvantem.
 - ▶ Na jednom jádru CPU běží pouze jedno vlákno orientované na CPU, které vyžaduje **$100xT$** času jednoho jádra pro dokončení výpočtu.
- **Ke kolika přepnutím kontextu dojde než se vlákno ukončí?**

Příklad 3: Prioritní RR plánování

- **Předpokládejme následující parametry systému.**
 - ▶ Používá se dynamická priorita s proměnným časovým kvantem.
 - ★ Při spuštění mají vlákna nastavenou prioritu P_0 a časové kvantum T .
 - ★ Pokud vlákno s aktuální prioritou P_i **využije celé časové kvantum**
⇒ v následujícím běhu poběží s prioritou $P_i - 1$ a dvojnásobným časovým kvantem.
 - ★ Pokud vlákno s aktuální prioritou P_i **nevyužije celé časové kvantum**
⇒ v následujícím běhu poběží s prioritou $P_i + 1$ a polovičním časovým kvantem.
 - ▶ Na jednom jádru CPU běží pouze jedno vlákno orientované na CPU, které vyžaduje **$100 \times T$** času jednoho jádra pro dokončení výpočtu.

- **Ke kolika přepnutím kontextu dojde než se vlákno ukončí?**



- Kontext se přepne 7x, protože $100 = 1 + 2 + 4 + 8 + 16 + 32 + 37$.

Příklad: "Time Sharing" (TS) třída v Solarisu

- Třída TS představuje plánování s odnímáním s dynamickou prioritou a proměnným časovým kvantem.
- Celý algoritmus je definovaný tabulkou, kterou lze zobrazit/nastavit pomocí příkazu `dispadmin`.

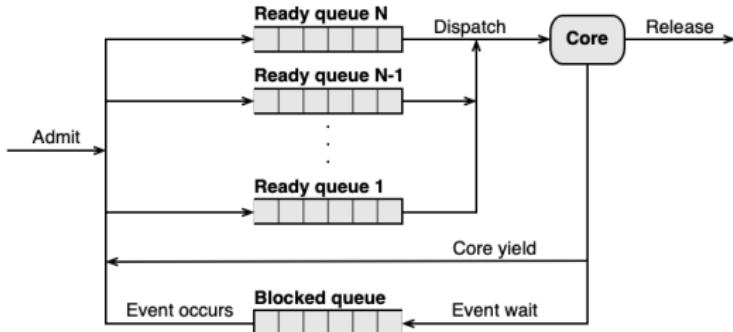
```
user@solaris:~> dispadmin -c TS -g
```

#	ts_quantum	ts_tqexp	ts_slpre	ts_maxwait	ts_lwait	PRIORITY	LEVEL
200	0	50	0	50	50	#	0
200	0	50	0	50	50	#	1
200	0	50	0	50	50	#	2
			...				
160	9	51	0	51	51	#	19
120	10	52	0	52	52	#	20
120	11	52	0	52	52	#	21
			...				
120	19	52	0	52	52	#	29
80	20	53	0	53	53	#	30
80	21	53	0	53	53	#	31
			...				
40	42	58	0	59	59	#	52
40	43	58	0	59	59	#	53
40	44	58	0	59	59	#	54
			...				
40	47	58	0	59	59	#	57
40	48	58	0	59	59	#	58
20	49	59	32000	59	59	#	59

Příklad: "Time Sharing" (TS) třída v Solarisu

- **Význam sloupečků ve výstupu příkazu `dispadmin`.**
 - ▶ Jedna řádka představuje parametry pro konkrétní hodnotu priority.
 - ★ **PRIORITY LEVEL:** hodnota priority.
 - ★ **ts_quantum:** velikost časového kvanta pro danou prioritu.
 - ★ **ts_tqexp:** nová hodnota priority pro vlákno, které využilo celé časové kvantum.
 - ★ **ts_slpret:** nová hodnota priority pro vlákno, které nevyužilo celé časové kvantum.
 - ★ **ts_maxwait:** počet sekund, které musí vlákno čekat v Ready frontě, než se mu nastaví nová priorita **ts_lwait** (hodnota 0 znamená, že po jedné sekundě se mu zvýší priorita).

- First-come-first-served (FCFS)



- ▶ Kooperativní plánování, ve kterém vlákna ve stavu "Ready" čekají v jedné/několika FIFO frontách na přidělení jádra CPU.
- ▶ Když běžící vlákno uvolní jádro CPU, první vlákno z Ready fronty s nejvyšším číslem bude pokračovat.
- ▶ Vlákna ve stavu "Blocked" čekají ve frontě Blocked.
- ▶ **Vlastnosti**
 - ★ Jednoduché na pochopení i implementování.
 - ★ Minimalizuje počet přepnutí kontextu.
 - ★ Zpomaluje vlákna orientovaná na V/V.

Příklad: "System" (SYS) třída v Solarisu

- Třída SYS představuje kooperativní plánování se statickou prioritou.
- Tato třída je vhodná pouze pro systémová vlákna, která krátce reagují na nějakou událost a je vhodné, aby reakci celou dokončila.
- Pouze kernel přiřazuje vlákna do této plánovací třídy.

```
user@solaris:~> ps -eLo class,pri,pid,lwp,comm | grep " *SYS "
  SYS  96      0      1 sched
  SYS  98      2      1 pageout
  SYS  97      2      2 pageout
  SYS  60      3      1 fsflush
  SYS  60      7      1 intrd
  SYS  60      8      1 vmtasks
  SYS  60    581      1 lockd_kproc
  SYS  60    581      2 lockd_kproc
  SYS  60  29436      1 nfs4cbd_kproc
  SYS  60  29436      2 nfs4cbd_kproc
15523
```

- V předchozím výpisu si můžeme např. všimnout procesu/vlákna `fsflush`, jehož úkolem je pravidelně ukládat změny ze skryté paměti (filesystem cache) do systému souborů (na disk).

Použité zdroje

- ① A. S. Tanenbaum, H. Bos: *Modern Operating Systems (4th edition)*, Pearson, 2014.
- ② W. Stallings: *Operating Systems: Internals and Design Principles (9th edition)*, Pearson, 2017.
- ③ A. Silberschatz, P. B. Galvin, G. Gagne: *Operating System Concepts (9th edition)*, Wiley, 2012.
- ④ R. McDougall, J. Mauro: *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture (2nd edition)*, Prentice Hall, 2006.

Operační systémy

Správa paměti – úvod.

Jan Trdlička



České vysoké učení technické v Praze, Fakulta informačních technologií
Katedra počítačových systémů

<https://courses.fit.cvut.cz/BI-OSY>

Obsah přednášky

1 Základní pojmy

- Kompilace a sestavení programu
- Zavedení programu do paměti

2 Fyzická organizace paměti

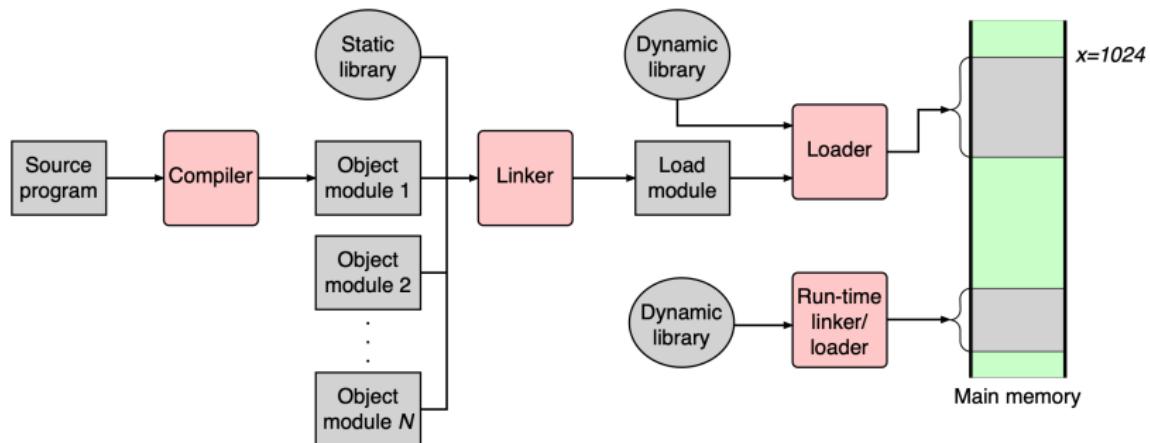
- Hierarchie pamětí
- Skrytá paměť (cache)

3 Logická organizace paměti

- Virtuální adresový prostor procesu (VAS)
- Implementace VAS
 - VAS identický s fyzickou pamětí
 - Dynamické oblasti
 - Stránkování

• Kompilace, sestavení a zavedení programu do paměti.

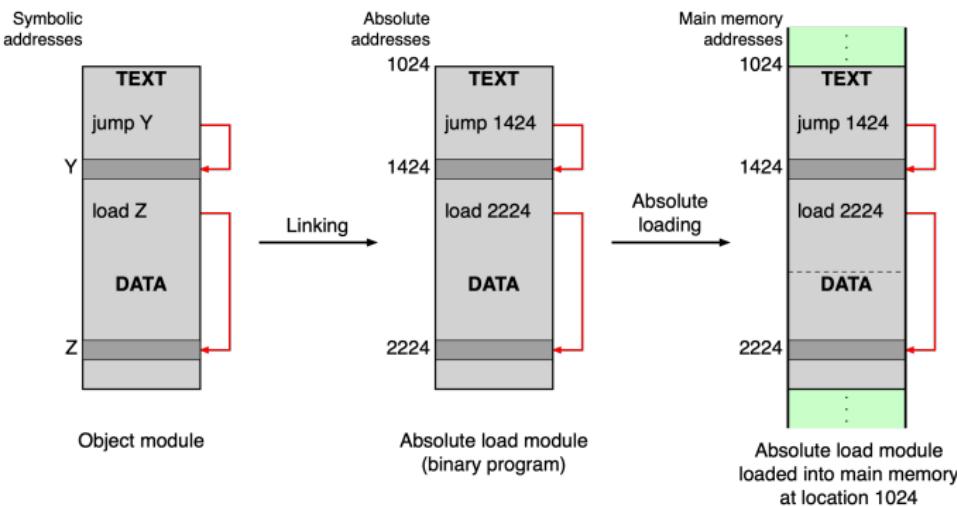
- ▶ **Object module:** množina funkcí, proměnných a metadat zkompilovaná do přemístitelného kódu v binárním formátu, který není přímo spustitelný.
- ▶ **Load module:** spustitelný binární program.
- ▶ Formát obou typů modulů je zavislý na OS (např. ELF, PE/PE32+).
- ▶ **Static library:** archiv objektových modulů ⇒ efektivnější sestavení.
- ▶ **Dynamic library:** "speciální" spustitelný binární program.



Zavedení programu do paměti (loading)

① Absolutní zavedení (absolute loading)

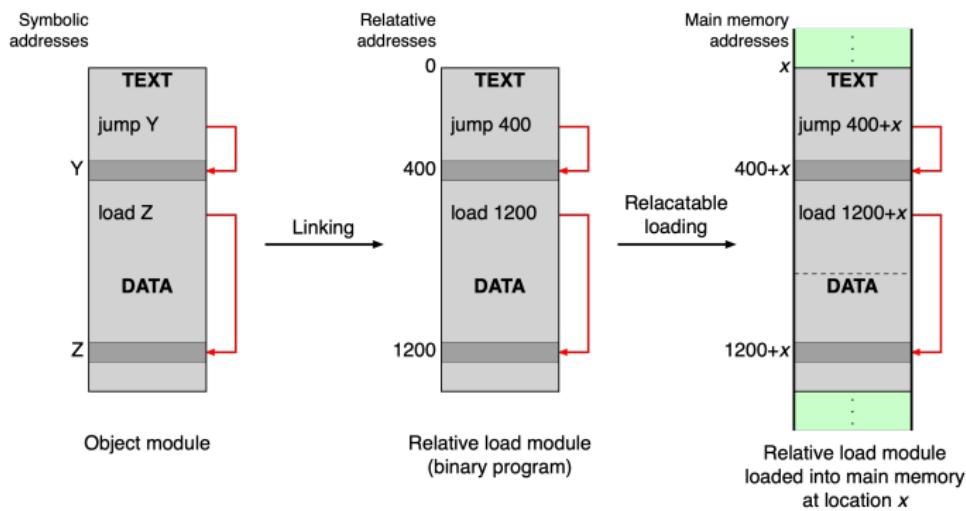
- ▶ Každý odkaz do paměti v bin. programu obsahuje **absolutní adresu**
⇒ program musí být zaveden vždy od dané fyzické adresy,
⇒ při sestavování musíme určit, kam bude zaveden a "Linker"
vypočítá absolutní fyzické adresy.



Zavedení programu do paměti (loading)

② Přemístitelné zavedení (relocatable loading)

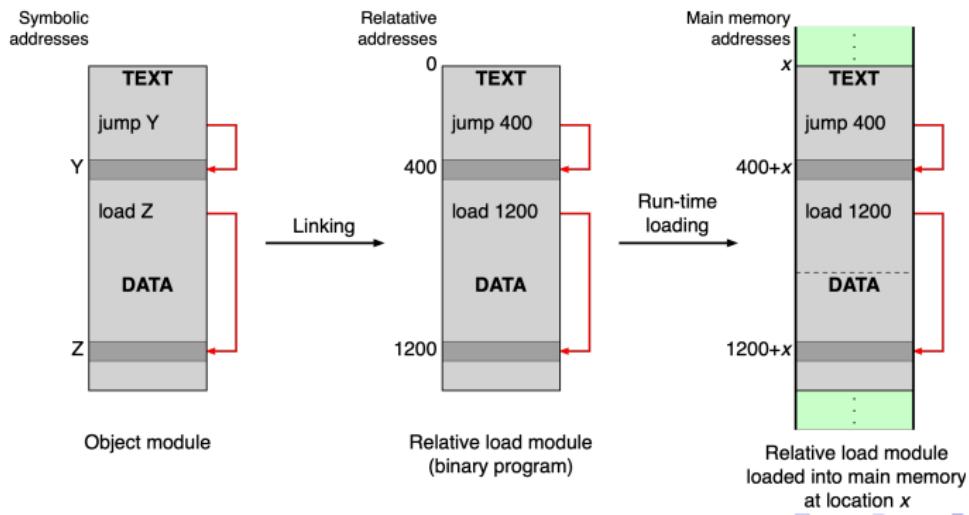
- ▶ Každý odkaz do paměti v bin. programu obsahuje **relativní adresu** (adresa vztažená k určitému bodu).
- ▶ "Loader" přepočítá relativní adresy na fyzické během zavedení programu do paměti.
- ▶ Informace o paměťových odkazech je uložena v "relocation dictionary" \Rightarrow rychlejší přepočítání.



Zavedení programu do paměti (loading)

③ Dynamic run-time loading

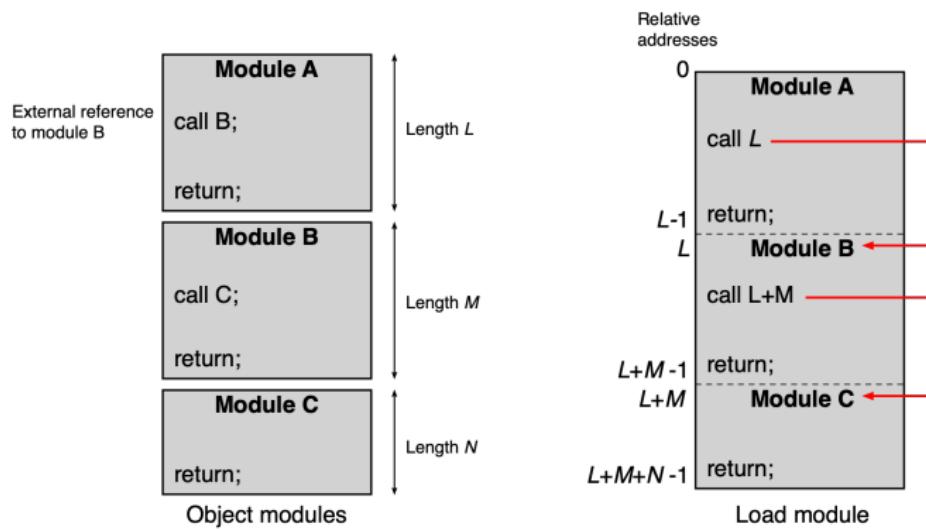
- ▶ Každý odkaz do paměti v bin. programu obsahuje relativní adresu.
- ▶ Program je zaveden do paměti s relativními adresami.
- ▶ Relativní adresa se přepočte na fyzickou teprve v okamžiku, kdy se přistupuje k instrukci/datům na této adrese.
- ▶ Přepočtení provádí hardware (MMU = Memory management unit) ve spolupráci s OS.



Sestavení programu (linking)

• Sestavení

- ▶ Sestavovací program (linker) vytvoří z jednoho/několika objektových modulů (object modules) jeden spustitelný modul (load module).
- ▶ Symbolické adresy objektových modulů musí být nahrazeny adresami definujícími umístění uvnitř spustitelného modulu vztaženými k jeho začátku.



Sestavení programu (linking)

1 Statické sestavení (static linking)

- ▶ "Linker" vytvoří z jednoho/více objektových modulů jeden **samostatný spustitelný modul** (load module) s absolutními adresami (absolute loading) nebo relativními adresami vztaženými k začátku modulu (relocatable loading/dynamic run-time loading).
- ▶ Při spuštění tohoto modulu (programu) stačí do paměti nahrát pouze tento modul, který obsahuje vše potřebné.

2 Dynamické sestavení (dynamic linking)

- ▶ "Linker" vytvoří **spustitelný modul obsahující odkazy na další moduly** (dynamické knihovny).
- ▶ Procesy sdílejí dynamické knihovny ⇒ v paměti je jedna instance každé knihovny.

a Load-time dynamic linking

- ★ Odkazy na další moduly se nahradí v okamžiku zavedení do paměti.
- ★ "Loader" musí zajistit, aby byly v paměti požadované knihovny.

b Run-time dynamic linking

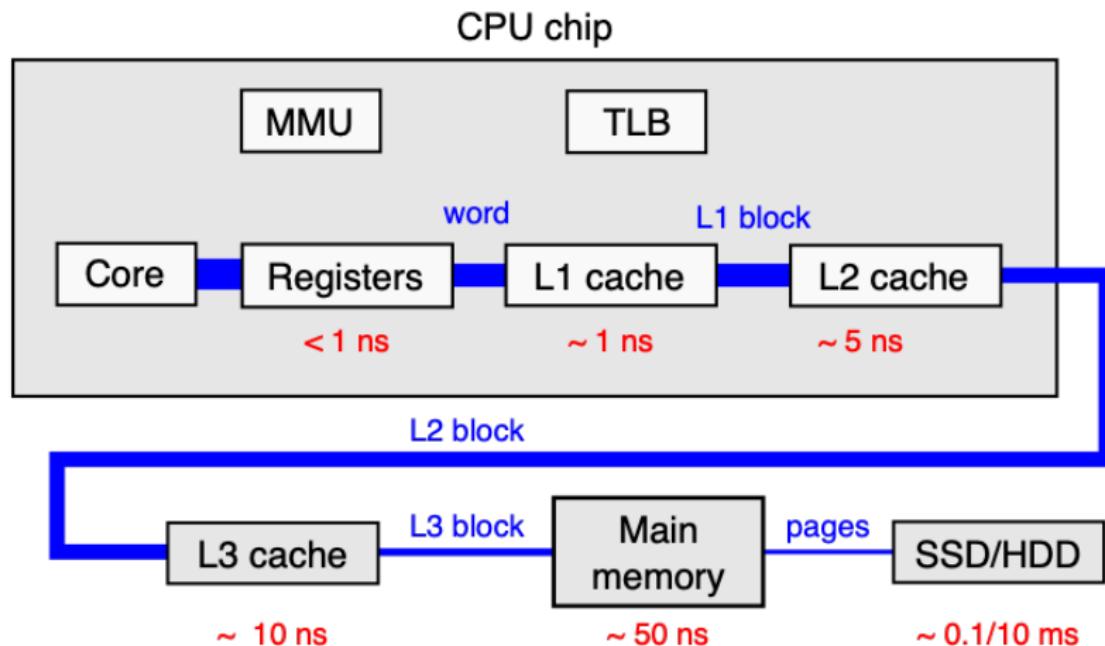
- ★ Odkazy na další moduly se nahradí v okamžiku volání (viz. funkce `dlopen()` v Unixu nebo `LoadLibraryA()` ve Windows).
- ★ "Run-time linker/loader zajistí nahrání požadovaných knihoven."

Fyzická organizace paměti

- Z důvodu rostoucího výkonu CPU na jedné straně a "pomalých" pamětí na druhé straně má fyzická paměť **hierarchickou organizaci**.
- Pomocí několika úrovní skrytých pamětí (cache) se minimalizuje průměrný čas přístupu k instrukcím/datům.
- Pro adresaci instrukcí/dat v hlavní paměti se používají fyzické adresy vztažené k začátku fyzické paměti.

Typ paměti	Velikost	Čas přístupu	Kdo spravuje
Registry CPU	1KB	<1ns	Překladač/programátor
L1 cache	64 KB	~1 ns	HW
L2 cache	256 KB	~5 ns	HW
L3 cache	4 MB	~10 ns	HW
Hlavní paměť	1 GB – 16 TB	~50 ns	OS
SSD	250 GB – 4TB	~0.1 ms	OS/procesy
HDD	500 GB – 16 TB	~10 ms	OS/procesy
Flash paměť	4 GB – 4 TB	~100 ms	OS/procesy
Páska	500 GB – 16 TB	>1s	OS/procesy

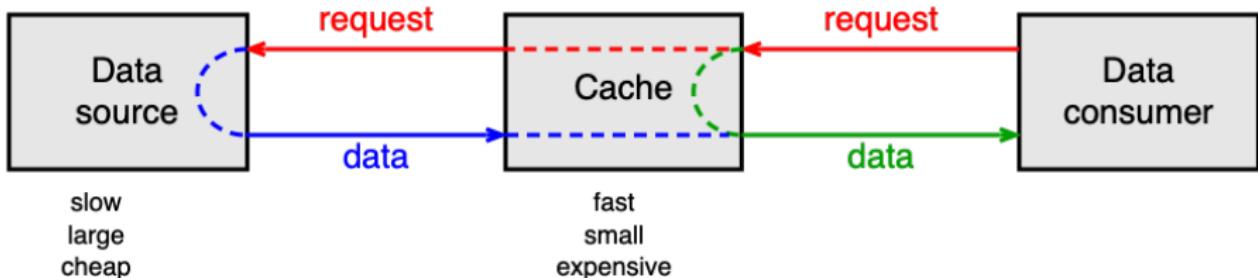
Fyzická organizace paměti



MMU = memory management unit

TLB = translation lookaside buffer

Skrytá paměť (cache)



- Skrytá paměť umožňuje **minimalizovat průměrnou dobu přístupu** k datům a je založena na následujících principech.
 - ▶ **Časová lokalita:** k datům, ke kterým se přistupovalo nedávno, se bude přistupovat i v blízké budoucnosti (skrytá paměť je "malá" ⇒ obsahuje pouze nedávno používaná data).
 - ▶ **Prostorová lokalita:** k datům, která jsou v okolí aktuálně používaných dat, se bude přistupovat v blízké budoucnosti (do skryté paměti se mohou načítat požadovaná data včetně okolních).
- **Příklady použití skryté paměti**
 - ▶ Všude, kde platí aspoň jeden z předchozích principů.
 - ▶ CPU (L1, L2 a L3 cache), disky, RAID, systémy souborů.

Skrytá paměť (cache)

• Výkonnostní parametry

- ▶ Cache hit count(n_h): počet případů, kdy data byla ve skryté paměti,
- ▶ Hit time (t_h): čas přístupu k datům ve skryté paměti,
- ▶ Cache miss count(n_m): počet případů, kdy data nebyla ve skryté paměti,
- ▶ Miss penalty (t_m): čas přístupu k datům ve zdroji dat (data source),
- ▶ Cache reference: celkový počet přístupů k datům $n_r = n_h + n_m$,
- ▶ Cache Hit Ratio: $r_h = \frac{n_h}{n_r} = \frac{n_h}{n_h+n_m}$,
- ▶ Average Access Time: $t_{avg} = t_h + (1 - r_h) \times t_m$.

• Příklad

- ▶ Předpokládejme následující parametry
 - ★ L1 cache: $t_h = 1$ ns,
 - ★ hlavní paměť: $t_m = 50$ ns.
- ▶ Jaký bude t_{avg} pro různě velké r_h ?
 - a) $r_h = 1.00: \quad t_{avg} = 1 + (1 - 1.00) \times 50 = 1$ ns,
 - b) $r_h = 0.90: \quad t_{avg} = 1 + (1 - 0.90) \times 50 = 6$ ns,
 - c) $r_h = 0.80: \quad t_{avg} = 1 + (1 - 0.80) \times 50 = 11$ ns,
 - d) $r_h = 0.00: \quad t_{avg} = 1 + (1 - 0.00) \times 50 = 51$ ns.

Skrytá paměť (cache)

• CPU

- ▶ Některé procesory umožňují monitorování svého výkonu pomocí programovatelných hardwarových čítačů, které se inkrementují při výskytu dané události (např. cache reference, cache miss, ...).
- ▶ Tyto informace lze zjistit v OS pomocí příslušných příkazů/aplikací (např. cpustat, cputrack, DTrace, SystemTap,...)
 - ⇒ můžeme zjistit, jak se systém/aplikace chová,
 - ⇒ můžeme opravit chyby v aplikaci (např. změnit přístup k datům).

• Příklad

- ▶ Pomocí následujícího příkazu můžeme zjistit "cache reference" (EC_ref) a "cache miss" (EC_misses) skrytých pamětí L3 na dvou instalovaných CPU v následujících 3 sekundách.

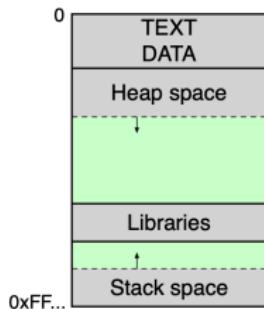
```
root@solaris:~ > cpustat -c EC_ref,EC_misses 1 3
```

time	cpu	event	pic0	pic1
1.008	0	tick	69284	1647
1.008	1	tick	43284	1175
2.008	0	tick	179576	1834
2.008	1	tick	202022	12046
3.008	0	tick	93262	384
3.008	1	tick	63649	1118
3.008	2	total	651077	18204



Virtuální adresový prostor procesu (VAS)

- Každý proces má svůj VAS, který se mapuje do fyzické paměti.
- Jakým způsobem bude VAS procesu mapován do fyzické paměti závisí na konkrétní architektuře CPU a použitém OS.
- **VAS typicky obsahuje oblasti pro následující struktury**



- ▶ program (TEXT),
- ▶ globální proměnné (DATA),
- ▶ halda (Heap space),
- ▶ knihovny (Libraries),
- ▶ zásobník/zásobníky (Stack space)

- Při rozmístění struktur ve VAS a při alokaci fyzické paměti je nutné pamatovat na to, že **některé struktury (halda, zásobník)** mění svojí **velikost** během existence procesu.
- Některé struktury ve VAS mohou být **sdílené** mezi procesy (TEXT, knihovny,...).
- **Address space layout randomization (ASLR)**: z důvodu bezpečnosti (ochrana např. proti útoku typu buffer-overflow, ...) mohou být struktury ve VAS umisťovány na náhodné adresy.

- VAS může být implementován různými způsoby.
- Podle počtu segmentů VAS (jeden/více), podle počtu procesů v fyzické paměti, a podle způsobu alokace fyzické paměti jednotlivým procesům (souvislé/nesouvislé oblasti) můžeme rozlišit následující implementace.

1 VAS jako jeden jednorozměrný logický prostor (segment)

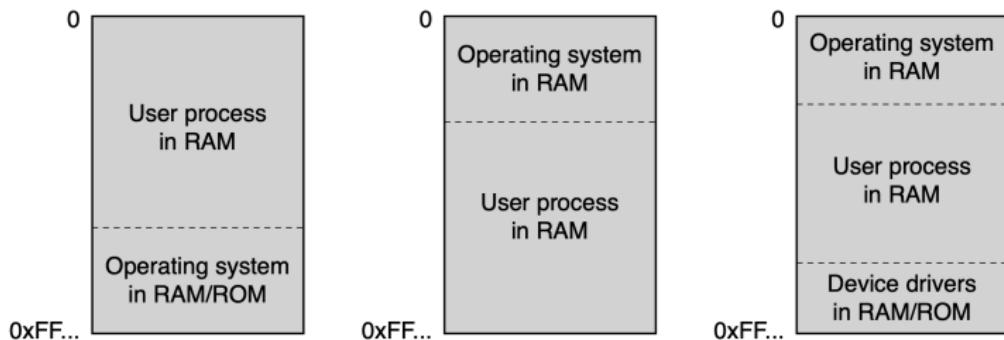
- ▶ Ve fyzické paměti bude pouze jeden proces.
 - a VAS identický s fyzickou pamětí: VAS je jednorozměrný prostor alokováný jako souvislá oblast ve fyzické paměti.
- ▶ Ve fyzické paměti může být současně více procesů.
 - b Dynamické oblasti: VAS alokován jako souvislá oblast ve fyzické paměti.
 - c Stránkování: VAS alokován jako množina stejně velkých oblastí (stránek) fyzické paměti.

2 VAS jako několik jednorozměrných segmentů ⇒ segmentace

- ▶ Ve fyzické paměti může být současně více procesů a VAS je složen z několika segmentů (jednorozměrný logický prostor).
 - a Dynamické oblasti: VAS alokován jako několik segmentů, kde každý segment je souvislá oblast ve fyzické paměti.
 - b Stránkování: VAS alokován jako několik segmentů, kde každý segment je množina stejně velkých oblastí (stránek) fyzické paměti.

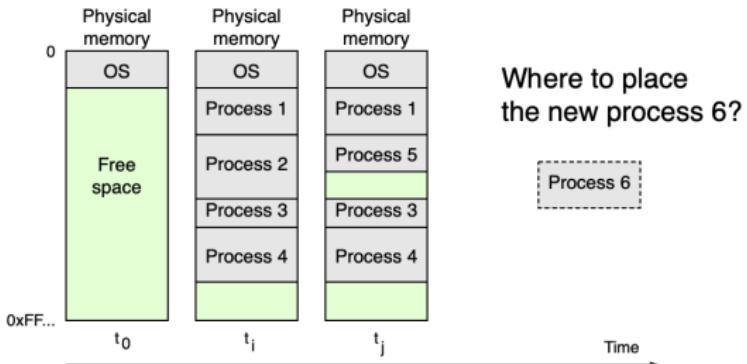
Implementace VAS – VAS identický s fyzickou pamětí

- Nejjednodušší implementace, která byla/je typická pro "main-frame" počítače (před 1960), mini počítače (před 1970), osobní počítače (před 1980) a pro některé současné jednoduché vestavěné (embedded) systémy.
- **Ve fyzické paměti se nachází pouze OS a jeden uživatelský proces**, který je nahrán vždy od stejné adresy (absolutní zavedení).
- Umístění OS a uživatelského procesu se může lišit pro různé architektury.



Implementace VAS – Dynamické oblasti

- Jeden ze způsobů, jak mapovat VAS do fyzické paměti, je alokovat jednu souvislou oblast fyzické paměti pro celý VAS.
- Tento způsob je označovaný jako "alokování pomocí dynamických oblastí" (Dynamic partitioning) a byl používaný v "main-frame" počítačích.
- Ve fyzické paměti je OS a typicky několik uživatelských procesů.



● Problém s fragmentací fyzické paměti

- Externí fragmentace: po určitém čase je volná paměť reprezentována příliš malými oblastmi, do kterých se již nevezdou nově vznikající procesy.
- Interní fragmentace: VAS obsahuje volnou paměť, do které se může rozpínat halda a zásobníky vláken.

- Při této alokaci paměti vznikají problémy, jejichž řešení vedlo ke vzniku virtuální paměti, stránkování a segmentace.

Implementace VAS – Dynamické oblasti

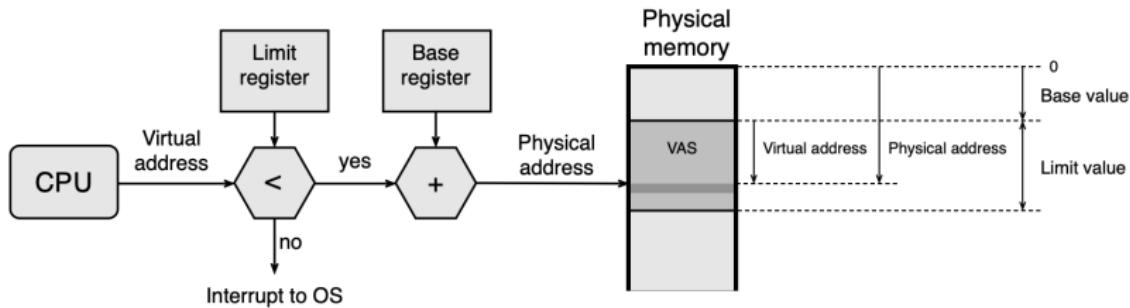
● Odkládání procesů (swapping)

- ▶ Technika, kdy při nedostatku paměti, se "vhodné" procesy (např. ve stavu "Blocked") dočasně přesunou na disk a uvolněné místo se přidělí dalším procesům ⇒ řeší dočasný nedostatek fyzické paměti.

● Přepočítávání logických adres na fyzické

- ▶ Relocatable loading.
- ▶ Dynamic run-time loading.

- ★ Řešení pomocí dvou hardwarových registrů: **Base registr** obsahuje fyzickou adresu začátku oblasti (relocation) a **Limit register** obsahuje velikost oblasti (oba registry současně zajišťují ochranu VAS).

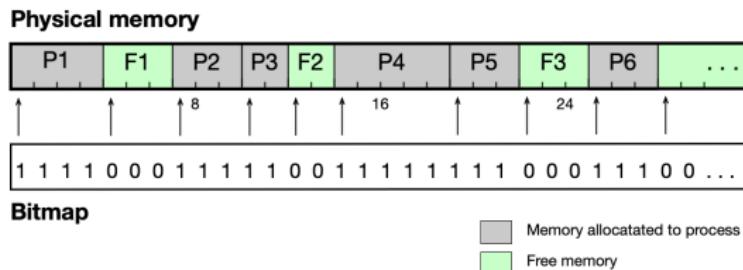


• Správa volné paměti

- ▶ Jedním z úkolů OS při správě paměti je udržování informace o volných oblastech fyzické paměti a jejich velikostech. K tomuto účelu lze použít bitovou mapu nebo zřetezené seznamy.

• Bitová mapa

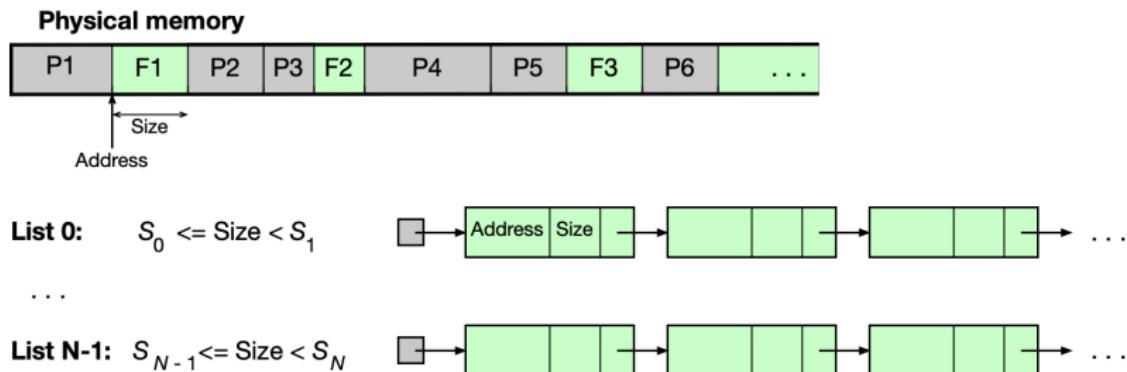
- ▶ Můžeme si představit, že fyzická paměť je složená z malých alokačních jednotek stejné velikosti. Potom v bitové mapě bude jeden bit (0=volná/1=alokovaná) reprezentovat jednu alokační jednotku.
- ▶ **Alokace paměti:** nalezení řetězce nul požadované délky \Rightarrow pomalé.
- ▶ **Uvolnění paměti:** změna příslušného řetězce jedniček na nuly.



Implementace VAS – Dynamické oblasti

• Zřetězené seznamy

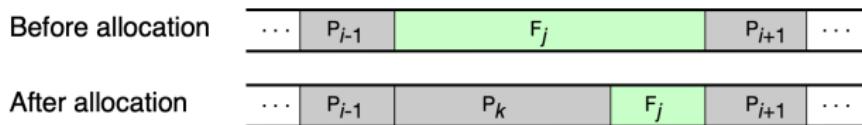
- ▶ Informaci o volných oblastech budeme uchovávat v N zřetězených seznamech L_i , $1 \leq i \leq N$.
- ▶ Každý seznam L_i bude obsahovat informace pouze o oblastech určitých velikostí. Nechť jsou definovány hodnoty $S_0 < \dots < S_N$. V seznamu L_i budou informace pouze o volných oblastech, pro jejichž velikost S platí $S_i \leq S < S_{i+1}$.



Implementace VAS – Dynamické oblasti

• Zřetězené seznamy

- ▶ **Alokace paměti:** Pokud chceme alokovat paměť velikosti S , pak použijeme první volnou oblast z příslušného seznamu L_i (neprázdný seznam s nejnižším i , kde $S_i \geq S$), ze které jednu část alokujeme procesu a informaci o druhé volné části (pokud bude existovat) přidáme do příslušného seznamu. ⇒ **rychlé**.



- ▶ **Uvolnění paměti:** Při uvolnění paměti musíme zjistit, zda před nebo za touto pamětí není volná oblast, a pokud ano, pak tyto oblasti musíme sloučit do jedné větší volné oblasti a umístit do příslušného seznamu ⇒ **pomalé**.

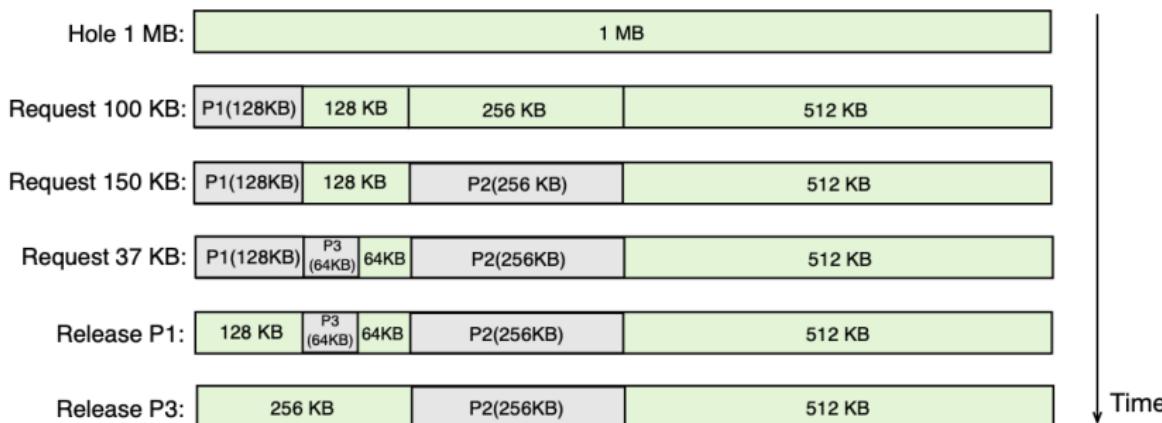


• Buddy systém

- ▶ Varianta zřetězených seznamů, ve kterých volné oblasti mají pouze velikosti, které jsou mocninou dvou. To umožní rychleji slučovat uvolněnou paměť.
- ▶ Alokace paměti
 - ★ Předpokládejme, že na počátku je celá paměť volná a má velikost 2^M .
 - ★ Pokud nový proces požaduje paměť o velikost S , pro kterou platí $S = 2^k$, potom mu přidělíme celou oblast 2^M .
 - ★ Jinak oblast 2^M budeme rekurzivně dělit na poloviny (vždy jednu z polovin), dokud nezískáme oblast 2^i , pro kterou $2^{i-1} < S \leq 2^i$ a tu přidělíme procesu.
 - ★ Nově vzniklé volné oblasti přidáme do příslušných seznamů ⇒ rychlé.
- ▶ Uvolnění paměti
 - ★ Opačný postup než při alokaci.
 - ★ Rekurzivně slučujeme odpovídající poloviny dokud to jde (dokud existují) ⇒ rychlé.

• Příklad: Buddy systém

- ▶ Na počátku je paměť o velikosti 1M je prázdná.
- ▶ Postupně jsou vytvořeny procesy P1, P2, P3, které vyžadují paměť o velikostech minimálně 100KB, 150KB a 37KB.
- ▶ Na závěr se ukončí proces P1 a P3.



• Problémy

- ▶ Fyzická paměť je po určitém čase fragmentovaná (obsahuje mnoho malých volných oblastí)
⇒ problém najít dostatečně velkou souvislou oblast pro nový proces.

• Řešení

- ▶ VAS budeme alokovat jako množinu malých oblastí
⇒ odpadne problém s nalezením volné velké souvislé oblasti,
⇒ stránkování.

- **Fyzická paměť (hlavní paměť)**

- ▶ Představme si, že je rozdělená na úseky stejné velikosti (např. 4KB) nazývané **rámce (frames)**.

- **VAS**

- ▶ Představme si, že je rozdělený na úseky stejné velikosti nazývané **stránky (pages)**.

- **Velikost rámce a stránky je stejná.**

- **Jednotlivé stránky VAS se nahrávají do volných rámců fyzické paměti.**

- **OS si musí pamatovat**

- ▶ rámce přidělené jednotlivým procesům (např. pomocí tabulky stránek,...),
 - ▶ volné rámce.

Implementace VAS – Stránkování

Frame number	Main memory
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	

Sixteen available frames

Main memory
A.0
A.1
A.2
A.3
A.4
A.5
B.0
B.1
B.2
C.0
C.1
C.2
C.3

Load processes A,B,C

Main memory
A.0
A.1
A.2
A.3
A.4
A.5
D.0
D.1
D.2
C.0
C.1
C.2
C.3
D.3

Swap process B
and load process D

0	0
1	1
2	2
3	3
4	4
5	5

Process A
page table

0	---
1	---
2	---

Process B
page table

0	9
1	10
2	11
3	12

Process C
page table

0	6
1	7
2	8
3	13

Process D
page table

14
15

Free frame
list

Použité zdroje

- ① A. S. Tanenbaum, H. Bos: *Modern Operating Systems (4th edition)*, Pearson, 2014.
- ② W. Stallings: *Operating Systems: Internals and Design Principles (9th edition)*, Pearson, 2017.
- ③ A. Silberschatz, P. B. Galvin, G. Gagne: *Operating System Concepts (9th edition)*, Wiley, 2012.

Operační systémy

Virtuální paměť, stránkování.

Jan Trdlička



České vysoké učení technické v Praze, Fakulta informačních technologií
Katedra počítačových systémů

<https://courses.fit.cvut.cz/BI-OSY>

Obsah přednášky

- 1 Virtuální paměť
- 2 Virtuální paměť se stránkováním
- 3 Překlad virtuálních adres na fyzické
 - Memory management unit (MMU)
 - Tabulka stránek
 - Víceúrovňová tabulka stránek
 - Invertovaná tabulka stránek
 - Translation lookaside buffer (TLB)

• Příklad

- ▶ Ve 32-bitovém OS, jeden 32-bitový proces může adresovat až 2^{32} B = 4 GB.
- ▶ V 64-bitovém OS, jeden 64-bitový proces může teoreticky adresovat až 2^{64} B = 16 EB (v praxi se zatím nevyužívá všech 64 bitů pro adresaci VAS).
- ▶ Většina procesů ale reálně používá pouze zlomek prostoru ze svého VAS.

• Problém

- ▶ VAS jednoho procesu může být větší než instalovaná fyzická paměť systému.
- ▶ OS umožňuje současně spustit až tisíce procesů \Rightarrow součet velikostí jednotlivých VAS je opět větší než instalovaná fyzická paměť systému.

• Řešení: Virtuální paměť

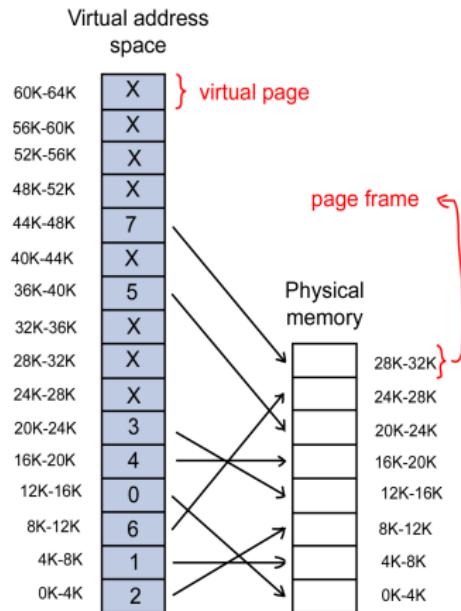
- ▶ VAS každého procesu je automaticky (pomocí HW/OS) rozdělen na menší kousky a ve fyzické paměti musí být pouze kousky aktuálně používané, zbytek používaného VAS je na disku.

Virtuální paměť se stránkováním

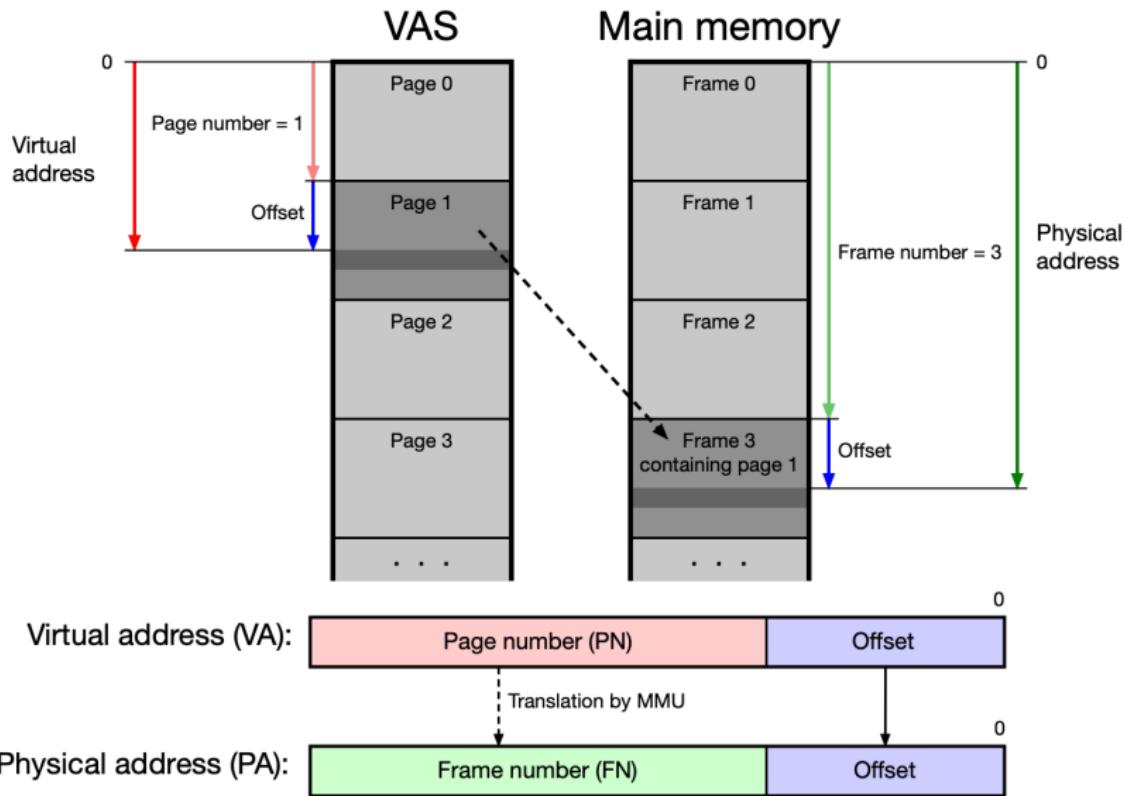
- Většinou je virtuální paměť kombinována se stránkováním.

• Princip

- ▶ Proces používá adresy, kterým se říká **virtuální adresy** a které adresují **virtuální adresový prostor**.
- ▶ VAS je rozdělen na stejně velké souvislé oblasti nazývané **virtuální stránky** (virtual pages). V závislosti na architektuře CPU je minimální velikost 4 KB (Intel), 8KB (Sparc).
- ▶ Korespondující úseky ve fyzické paměti stejné velikosti jsou nazývány **rámce stránek** (page frames).
- ▶ V hlavní paměti musí být aspoň stránky aktuálně používané.

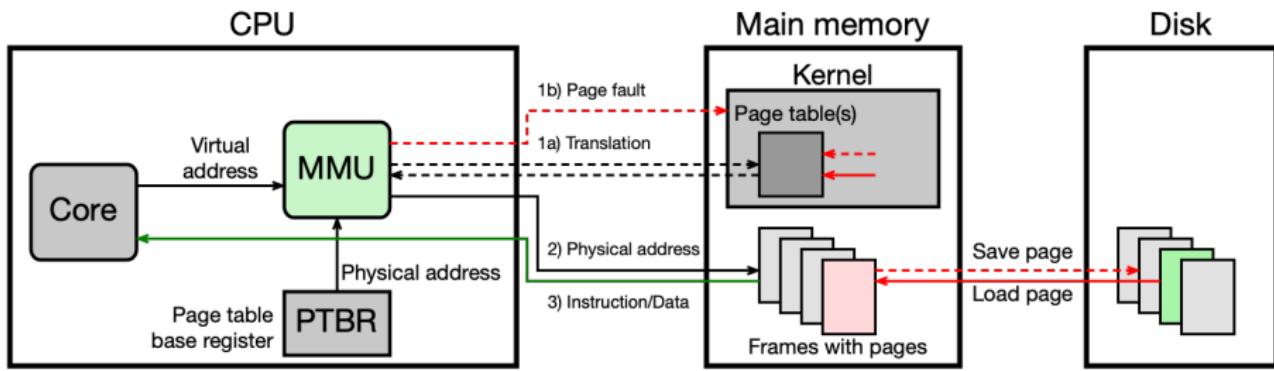


Struktura virtální a fyzické adresy



Memory management unit (MMU)

- Proces má ilusi, že celý jeho VAS je v hlavní paměti a pro adresaci instrukcí/dat používá virtuální adresy (vztažené k začátku VAS).
- Tuto ilusi a překlad adres zajišťuje hardware, který se nazývá "**Memory management unit**" (**MMU**), ve spolupráci s OS.
- **Výpadek stránky (Page fault)**
 - ▶ Pokud není virtuální stránka v hlavní paměti, MMU způsobí přerušení a "požádá" OS o nahrání příslušné stránky do fyzické paměti.
 - ▶ OS nejdříve najde vhodný rámeček fyzické paměti (pokud je nutné, uloží jeho obsah na disk), a pak do něj nahraje požadovanou virtuální stránku z disku nebo v něm "vytvoří" novou stránku (zá sobník, heap).



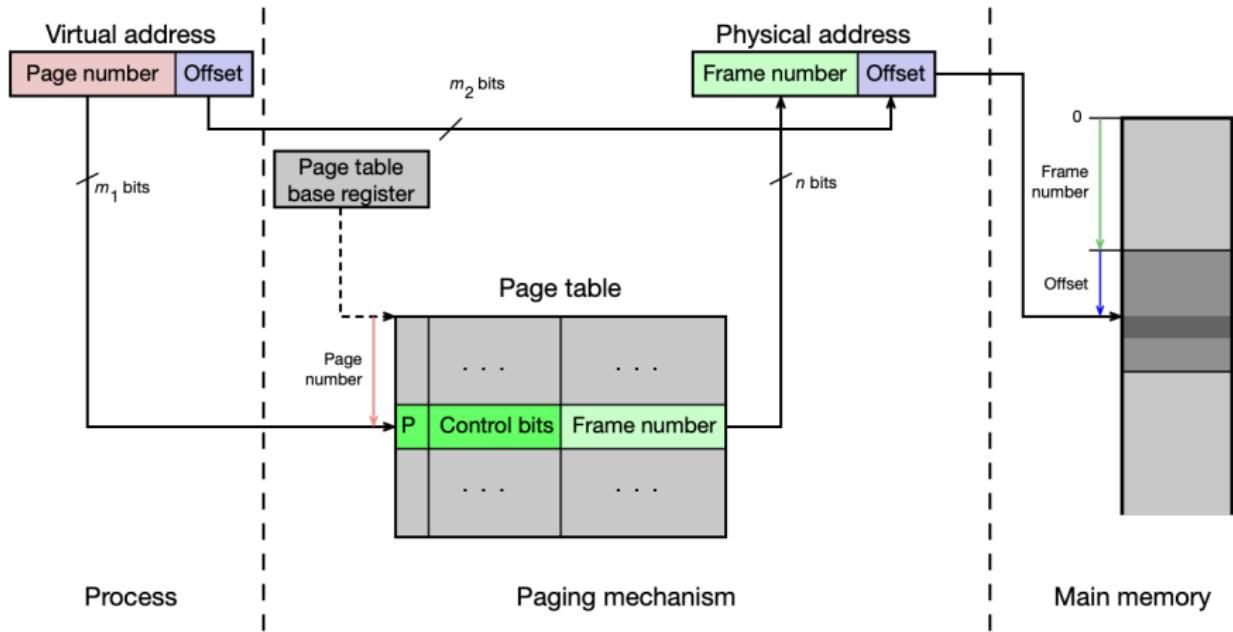
Překlad virtuálních adres na fyzické

- OS si musí udržovat informaci o tom, do kterých rámců fyzické paměti se namapovaly jednotlivé stránky VAS jednotlivých procesů, a které rámce fyzické paměti jsou zatím volné.
- Tuto informaci si OS udržuje v následujících strukturách, které jsou závislé na ISA použitého procesoru,
 - ▶ tabulka stránek (page table),
 - ▶ víceúrovňová tabulka stránek (multilevel page table),
 - ▶ invertovaná tabulka stránek (inverted page table).
- Pro urychlení překladu pak MMU využívá "**Translation lookaside buffer**" (TLB), ve kterém jsou informace o naposledy překládaných virtuálních adresách.

Tabulka stránek

- Tabulka obsahuje pro každou stránku VAS daného procesu jednu řádku, ve které jsou uložené následující informace
 - ▶ **číslo rámce**, do kterého se tato stránka namapovala,
 - ▶ **kontrolní bity**
 - ★ Present bit (P): informace, zda stránka je v hlavní paměti,
 - ★ Reference bit (R): informace, zda se ke stránce přistupovalo,
 - ★ Modify bit (M): informace, zda byl obsah stránky modifikován,
 - ★ Přístupová práva
 - ★ Cache disable/enabled: zda jsou registry periferií mapovány přímo do paměti,
 - ★ Read/write (R/W): informace, zda lze stránku modifikovat,
 - ★ User/supervisor (U/S): informace, zda lze na stránku přistupovat v uživatelském modu, ...
- Číslo stránky (page number) funguje jako **index** do této tabulky.
- OS si musí udržovat **pro každý proces jednu tabulku**.

Tabulka stránek: Překlad virtuální adresy



Tabulka stránek

• Příklad

- ▶ Na 32-bitovém CPU, které podporuje pouze **4KB stránky/rámce**, je nainstalovaný 32-bitový OS, ve kterém běží 32-bitový proces.
- ▶ Předpokládejme, že proces bude alokovat ve svém VAS pouze následující datové struktury
 - ★ TEXT a DATA: **4 MB** na nejnižších virtuálních adresách,
 - ★ halda: **4 MB** na následujících virtuálních adresách,
 - ★ zásobník: **4 MB** na nejvyšších virtuálních adresách.

Tabulka stránek

• Příklad

- ▶ Na 32-bitovém CPU, které podporuje pouze 4KB stránky/rámce, je nainstalovaný 32-bitový OS, ve kterém běží 32-bitový proces.
- ▶ Předpokládejme, že proces bude alokovat ve svém VAS pouze následující datové struktury
 - ★ TEXT a DATA: 4 MB na nejnižších virtuálních adresách,
 - ★ halda: 4 MB na následujících virtuálních adresách,
 - ★ zásobník: 4 MB na nejvyšších virtuálních adresách.

• Jaká bude struktura virtuální a fyzické adresy?

- ▶ Virtuální adresa (32 bitů): číslo stránky (20 bitů) + offset (12 bitů).
- ▶ Fyzická adresa (32 bitů): číslo rámce (20 bitů) + offset (12 bitů).

• Kolik řádek bude mít tabulka stránek?

- ▶ Pro každou stránku musí být jedna řádku $\Rightarrow 2^{20}$ řádek.

• Kolik místa zabírá jedna řádka?

- ▶ Číslo rámce (20 bitů) + kontrolní byty (P+R+M+...) \Rightarrow zaokrouhlíme na celé bytes/slova ~ 32 bitů = 4 B.

• Kolik místa zabírají všechny tabulky pokud na systému běží 2^7 podobných procesů?

- ▶ $2^7 \times [2^{20} \times 4] \text{ B} = 2^7 \times 4 \text{ MB} = 512 \text{ MB.}$

Tabulka stránek: Problémy

• **Problémy**

- ▶ Přestože proces používá pouze zlomek místa ze svého VAS, tak tabulka stránek obsahuje informaci i o nepoužitých stránkách.
- ▶ Pro každý proces musí OS držet jednu tabulku stránek.
⇒ plýtvání pamětí.

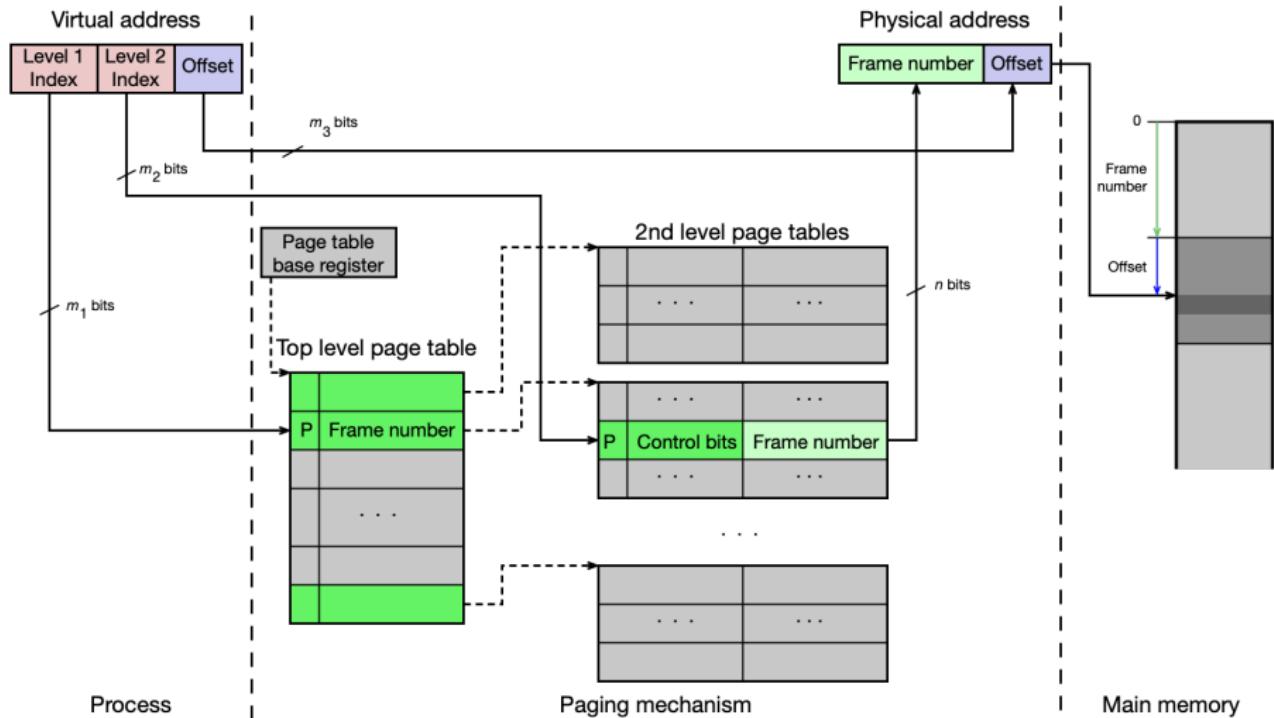
• **Řešení**

- ▶ Víceúrovňová tabulka stránek, která umožňuje se držet v paměti pouze informace o používaných stránkách.
⇒ u procesů, které alokují málo paměti, můžeme výrazně ušetřit.

Víceúrovňová tabulka stránek

- Pro n úrovňovou tabulku stránek platí
 - ▶ virtuální adresa se skládá z n indexů, které ukazují do tabulek jednotlivých úrovní, a offsetu,
 - ▶ fyzická adresa se skládá z čísla rámce a offsetu,
 - ▶ tabulky stránek úrovní $1, \dots, n-1$ obsahují "present bit" a číslo rámce, ve kterém je uložena/začíná tabulka následující úrovně,
 - ▶ tabulka stránek úrovně n obsahuje "present bit" a číslo rámce, ve kterém je uložena samotná virtuální stránka.
- V hlavní paměti je vždy "Top level tabulka" (tabulka úrovně 1).
- Tabulky ostatních úrovní v paměti být nemusí, pokud proces nepoužívá stránky z oblastí VAS, která tyto tabulky pomáhají adresovat ⇒ šetří se fyzická paměť.
- Za ušetřené místo však platíme pomalejším překladem
⇒ pro urychlení překladu se používá společně s TLB.

Dvouúrovňová tabulka stránek



Víceúrovňová tabulka stránek

● Příklad

- ▶ Na 32-bitovém CPU, které podporuje pouze 4KB stránky/rámce, je nainstalovaný 32-bitový OS, ve kterém běží 32-bitový proces. Systém používá dvouúrovňovou tabulku stránek a indexy do jednotlivých tabulek mají stejnou velikost.
- ▶ Předpokládejme, že proces bude alokovat ve svém VAS pouze následující datové struktury
 - ★ TEXT a DATA: 4 MB na nejnižších virtuálních adresách,
 - ★ halda: 4 MB na následujících virtuálních adresách,
 - ★ zásobník: 4 MB na nejvyšších virtuálních adresách.

Víceúrovňová tabulka stránek

● Příklad

- ▶ Na 32-bitovém CPU, které podporuje pouze 4KB stránky/rámce, je nainstalovaný 32-bitový OS, ve kterém běží 32-bitový proces. Systém používá dvouúrovňovou tabulku stránek a indexy do jednotlivých tabulek mají stejnou velikost.
- ▶ Předpokládejme, že proces bude alokovat ve svém VAS pouze následující datové struktury
 - ★ TEXT a DATA: 4 MB na nejnižších virtuálních adresách,
 - ★ halda: 4 MB na následujících virtuálních adresách,
 - ★ zásobník: 4 MB na nejvyšších virtuálních adresách.

● Jaká bude struktura virtuální a fyzické adresy?

- ▶ Virtuální adresa (32 bitů): level 1 index (10 bitů) + level 2 index (10 bitů) + offset (12 bitů).
- ▶ Fyzická adresa (32 bitů): číslo rámce (20 bitů) + offset (12 bitů).

● Kolik řádek bude mít top level tabulka? $\Rightarrow 2^{10}$ řádek.

● Kolik místa zabírá jedna řádka?

- ▶ Číslo rámce (20 bitů) + kontrolní bit (P) \Rightarrow zaokrouhlíme ~ 32 bitů = 4 B.

● Kolik řádek bude mít tabulka druhé úrovně? $\Rightarrow 2^{10}$ řádek.

● Kolik místa zabírá jedna řádka?

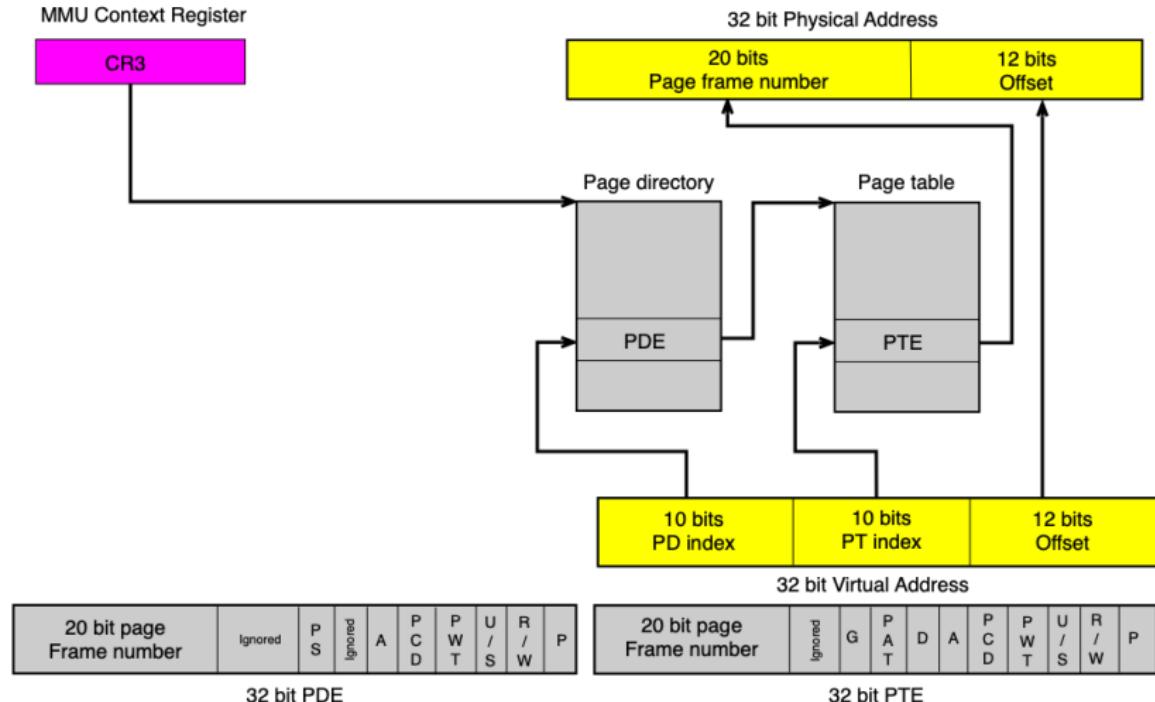
- ▶ Číslo rámce (20 bitů) + kontrolní byty (P,...) \Rightarrow zaokrouhlíme ~ 32 bitů = 4 B.

● Kolik místa zabírají všechny tabulky pokud na systému běží 2^7 podobných procesů?

- ▶ $2^7 \times [1 \times (2^{10} \times 4) + 3 \times (2^{10} \times 4)] \text{ B} = 2^7 \times 16 \text{ KB} = 2 \text{ MB.}$

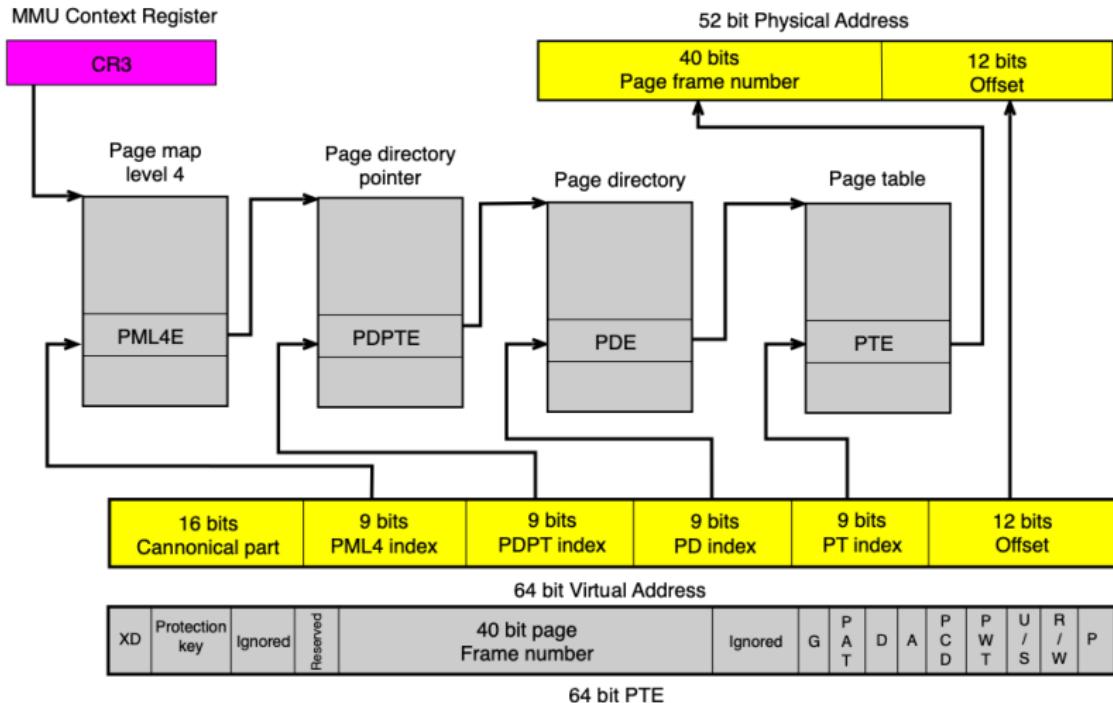
Víceúrovňová tabulka stránek

- Příklad dvouúrovňové tabulky stránek procesoru x86.



Víceúrovňová tabulka stránek

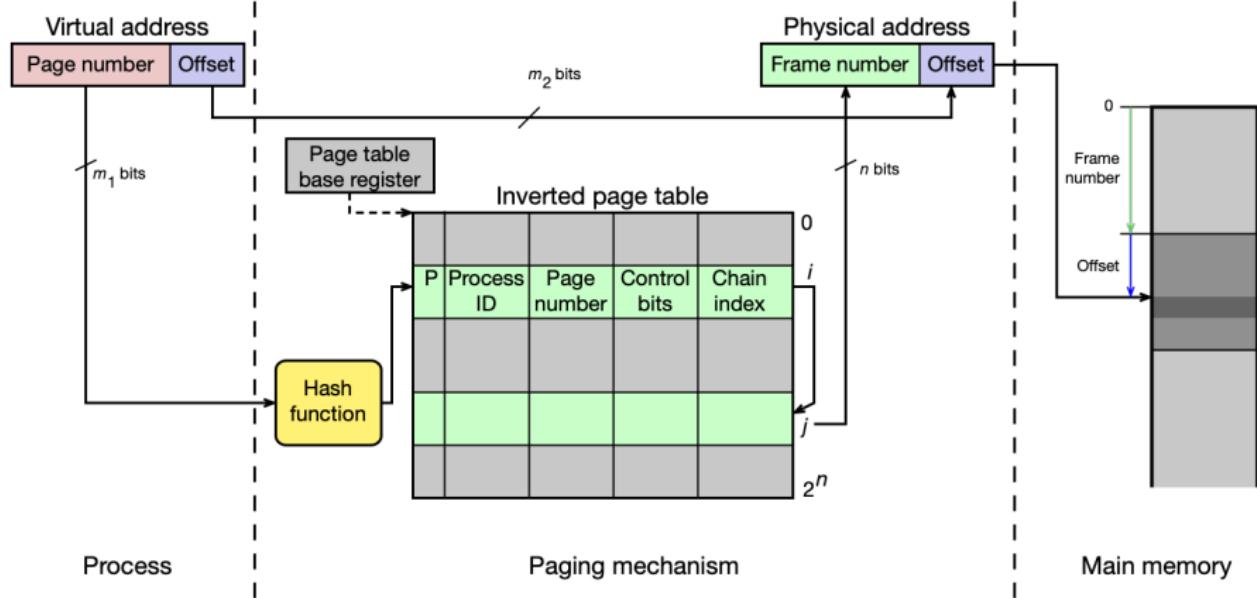
- Příklad čtyřúrovňové tabulky stránek procesoru x86-64.



Invertovaná tabulka stránek

- Tabulka obsahuje pro každý rámec fyzické paměti jednu řádku, ve které jsou uložené následující informace
 - ▶ **číslo stránky**, která je nahrána do tohoto rámce,
 - ▶ **číslo procesu**, do jehož VAS tato stránka patří,
 - ▶ **kontrolní bity**
 - ★ Present bit (P): informace, zda stránka je v hlavní paměti,
 - ★ Reference bit (R): informace, zda se ke stránce přistupovalo,
 - ★ Modify bit (M): informace, zda byl obsah stránky modifikován,
 - ★ Přístupová práva
 - ★ Cache disable/enabled: zda jsou registry periferií mapovány přímo do paměti, ...
 - ▶ **index zřetězení (chain)** .
- OS si musí udržovat **pouze jednu tabulku** pro celý systém.
- **Číslo stránky (page number)** se pomocí hašovací funkce přepočte na **index do tabulky**. Protože počet stránek je větší než počet rámců fyzické paměti, **několik různých stránek se může namapovat na stejnou řádku v tabulce** ⇒ proto ze používá technika zřetězení.

Invertovaná tabulka stránek



Invertovaná tabulka stránek

● Příklad

- ▶ Na 32-bitovém CPU, které podporuje pouze **4KB stránky/rámce**, je nainstalovaný 32-bitový OS, ve kterém běží 32-bitový proces. Systém používá invertovanou tabulku stránek a lze na něm vytvořit maximálně 2^{10} procesů.
- ▶ Předpokládejme, že proces bude alokovat ve svém VAS pouze následující datové struktury
 - ★ TEXT a DATA: **4 MB** na nejnižších virtuálních adresách,
 - ★ halda: **4 MB** na následujících virtuálních adresách,
 - ★ zásobník: **4 MB** na nejvyšších virtuálních adresách.

Invertovaná tabulka stránek

● Příklad

- ▶ Na 32-bitovém CPU, které podporuje pouze 4KB stránky/rámce, je nainstalovaný 32-bitový OS, ve kterém běží 32-bitový proces. Systém používá invertovanou tabulku stránek a lze na něm vytvořit maximálně 2^{10} procesů.
- ▶ Předpokládejme, že proces bude alokovat ve svém VAS pouze následující datové struktury
 - ★ TEXT a DATA: 4 MB na nejnižších virtuálních adresách,
 - ★ halda: 4 MB na následujících virtuálních adresách,
 - ★ zásobník: 4 MB na nejvyšších virtuálních adresách.

● Jaká bude struktura virtuální a fyzické adresy?

- ▶ Virtuální adresa (32 bitů): číslo stránky (20 bitů) + offset (12 bitů).
- ▶ Fyzická adresa (32 bitů): číslo rámce (20 bitů) + offset (12 bitů).

● Kolik řádek bude mít tabulka stránek?

- ▶ Pro každý rámec musí být jedna řádka $\Rightarrow 2^{20}$ řádek.

● Kolik místa zabírá jedna řádka?

- ▶ Číslo procesu (10) + číslo stránky (20 bitů) + kontrolní bity (P+R+M+...) + chain index (20) \Rightarrow zaokrouhlíme na celé bytes/slova ~ 64 bitů = 8 B.

● Kolik místa budou všechny tabulky zabírat pokud běží v systému 2^7 podobných procesů?

- ▶ $1 \times 2^{20} \times 8 \text{ B} = 8 \text{ MB.}$

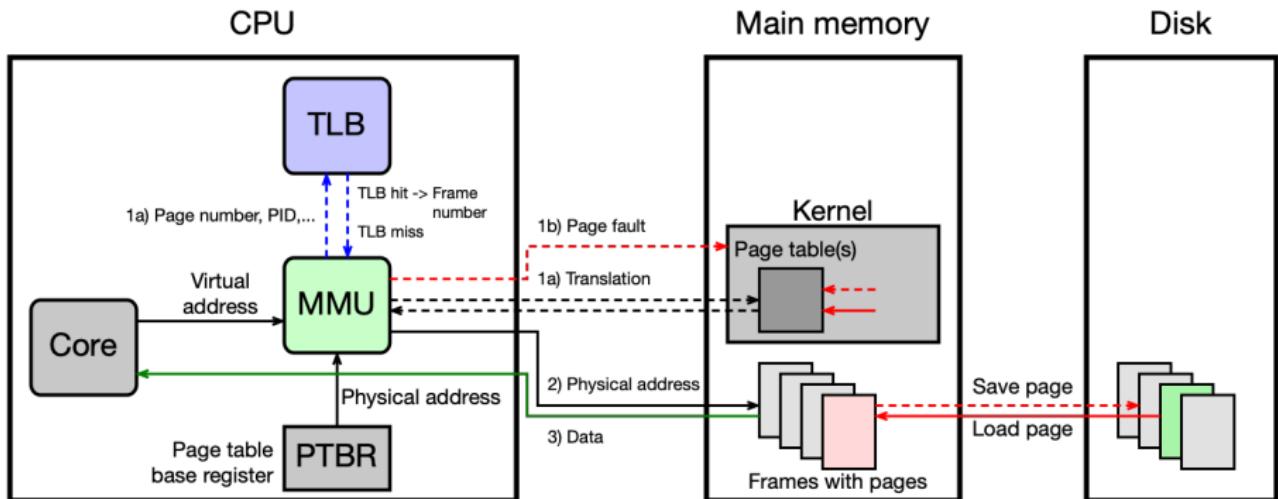
Invertovaná tabulka stránek

- Invertovaná tabulka stránek se používala/používá v procesorech PowerPC a UltraSPARC.
- Zabírá méně místa než tabulka stránek, ale hledání v této tabulce je pomalé. ⇒ pro urychlení překladu se používá společně s TLB.

Translation lookaside buffer (TLB)

- Za účelem urychlení překladu se v procesorech používá "Translation lookaside buffer" (TLB).
- TLB je v praxi implementovan jako *n*-way cache (paměť s omezeným stupněm asociativity).
- Obsahuje informace o posledně překládaných virtuálních adresách (číslo stránky–číslo rámec).
- Počet položek TLB je výrazně menší než počet virtuálních stránek/počet fyzických rámců.
- Položka TLB obsahuje
 - ▶ valid bit: zda je platná položka,
 - ▶ číslo stránky,
 - ▶ číslo rámce,
 - ▶ ASID: address space ID (identifikátor adresního prostoru),
 - ▶ control bits,...

Translation lookaside buffer (TLB)



Translation lookaside buffer (TLB)

- Moderní CPU cca od roku 2000 podporují současné používání různě velkých virtuální stránek/fyzický rámců.
- V některých OS (např. Solaris, AIX) je tato funkciálnita podporována implicitně, v jiných OS je nutné tuto vlastnost povolit.
- **Příklad**
 - ▶ Pomocí příkazu `pagesize` můžeme v Solarisu zjistit jaké velikosti stránek CPU podporuje (velikost je v B).

```
user@solaris:~ > pagesize -a
8192
65536
4194304
268435456
```

Translation lookaside buffer (TLB)

• Příklad

- ▶ Pomocí příkazu pmap můžeme v Solarisu zjistit jaké velikosti stránek jsou požívané ve VAS procesu s PID=7291.

```
user@solaris:~ > pmap -s 7291
7291:  bash
        Address      Bytes PgSz Mode      Mapped File
0000000100000000      960K  64K r-x---- /usr/bin/bash
00000001000F0000      40K   8K r-x---- /usr/bin/bash
00000001001FA000      48K   8K rwx---- /usr/bin/bash
0000000100300000      16K   8K rw---- /usr/bin/bash
0000000D1A2D6A000     24K   8K rw---- [ heap ]
0000000D1A2D70000     256K  64K rw---- [ heap ]
00007FA191B00000      192K  64K r-x---- /lib/sparcv9/libcurses.so.1
.
.
.
00007FA191F00000      64K   64K rwx---- [ anon ]
00007FA192000000      64K   64K rwx---- [ anon ]
00007FA19204C000      8K    8K rwxS--- [ anon ]
00007FA192100000      24K   8K rwx---- [ anon ]
00007FA192200000      384K  64K r-x---- /lib/sparcv9/libc.so.1
00007FA192380000      8K    8K r-x---- /lib/sparcv9/libc.so.1
00007FA192486000      64K   8K rwx---- /lib/sparcv9/libc.so.1
00007FA192496000      8K    8K rwx---- /lib/sparcv9/libc.so.1
00007FA192500000      64K   64K rw---- [ anon ]
00007FA192530000      8K    8K r--s--- [ anon ]
00007FA192600000      256K  64K r-x---- /lib/sparcv9/ld.so.1
00007FA192640000      16K   8K r-x---- /lib/sparcv9/ld.so.1
00007FA192744000      24K   8K rwx---- /lib/sparcv9/ld.so.1
FFFFFD097960000      24K   8K rw---- [ stack ]
total                  3848K
```



Translation lookaside buffer (TLB)

- Pokud aplikace používá/alokuje velkou část ze svého VAS, pak může docházet k velké frekvenci TLB miss (časté přepisování položek v TLB).
- CPU i OS nám umožňují sledovat jak využití TLB, tak i ostatních parametry "Memory managementu".
⇒ na základě těchto informací můžeme změnit nastavení OS/chování aplikací (např. pro některé oblasti VAS (třeba pro "Heap") můžeme nestavit používání stránek větších velikostí).

Translation lookaside buffer (TLB)

• Příklad

► Pomocí příkazu `trapstat -t` můžeme v Solarisu zjistit využití TLB.

- ★ u = user mode/k = kernel mode
- ★ itlb miss= instruction TLB miss / dtlb miss = data TLB miss
- ★ %tim = percentage of CPU time in miss handler
- ★ TSB = Translation Storage Buffer (variancia invertované tabulky stránek)

trapstat -t													
	cpu	m	itlb-miss	%tim	itsb-miss	%tim	dtlb-miss	%tim	dtsb-miss	%tim	%tim		
0	u	2571	0.3		0	0.0		10802	1.3		0	0.0	1.6
0	k	0	0.0		0	0.0		106420	13.4		184	0.1	13.6
1	u	3069	0.3		0	0.0		10983	1.2		100	0.0	1.6
1	k	27	0.0		0	0.0		106974	12.6		19	0.0	12.7
2	u	3033	0.3		0	0.0		11045	1.2		105	0.0	1.6
2	k	43	0.0		0	0.0		107842	12.7		108	0.0	12.8
3	u	2924	0.3		0	0.0		10380	1.2		121	0.0	1.6
3	k	54	0.0		0	0.0		102682	12.2		16	0.0	12.2
4	u	3064	0.3		0	0.0		10832	1.2		120	0.0	1.6
4	k	31	0.0		0	0.0		107977	13.0		236	0.1	13.1
ttl		14816	0.3		0	0.0		585937	14.1		1009	0.0	14.5

Použité zdroje

- ① A. S. Tanenbaum, H. Bos: *Modern Operating Systems (4th edition)*, Pearson, 2014.
- ② W. Stallings: *Operating Systems: Internals and Design Principles (9th edition)*, Pearson, 2017.
- ③ A. Silberschatz, P. B. Galvin, G. Gagne: *Operating System Concepts (9th edition)*, Wiley, 2012.
- ④ R. McDougall, J. Mauro: *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture (2nd edition)*, Prentice Hall, 2006.
- ⑤ Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4, 2021.

Operační systémy

Segmentace, algoritmy pro náhradu stránek,
návrh stránkovacích systémů.

Jan Trdlička



České vysoké učení technické v Praze, Fakulta informačních technologií
Katedra počítačových systémů

<https://courses.fit.cvut.cz/BI-OSY>

Obsah přednášky

1 Segmentace

- Jednorozměrný VAS x Segmentace
- Čistá segmentace
- Segmentace se stránkováním

2 Algoritmy pro náhradu stránek

- Optimální algoritmus
- NRU algoritmus (Not Recently Used)
- FIFO algoritmus (First-In First-Out)
- Clock algoritmus
- LRU algoritmus (Least Recent Used)
- Aging algoritmus

3 Návrh systémů se stránkováním

Jednorozměrný VAS x Segmentace

● Jednorozměrný VAS procesu

- ▶ Obsahuje datové struktury s různými vlastnostmi
 - ★ **velikost:** statická (TEXT, DATA) x dynamická (halda, zásobník),
 - ★ **typ přístupu:** privátní (zásobník,...) x sdílený (knihovny, sdílená paměť,...),
- ⇒ mapování těchto struktur do jednorozměrného VAS je umělé,
- ⇒ problémy při implementaci pomocí dynamických oblastí
 - ★ fragmentace fyzické paměti (problém s nalezením volné souvislé oblasti pro nový proces),
 - ★ rezervace místa ve VAS pro dat. struktury s dynamickou velikostí,
 - ★ sdílení struktur mezi více procesy.

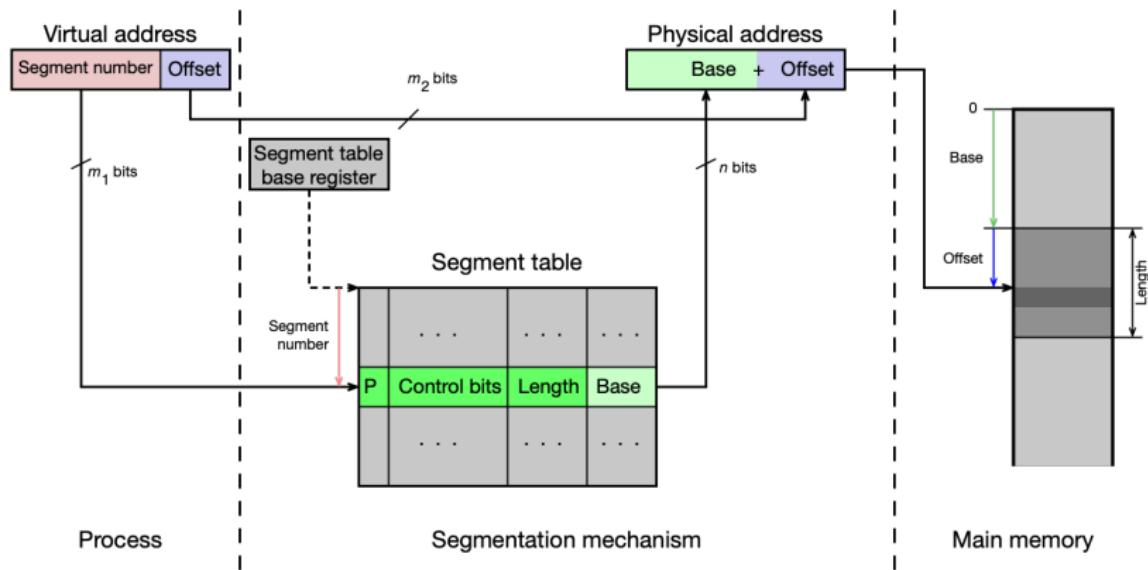
● Řešení

- ▶ **Stránkování:** řeší problémy v rámci jednorozměrného VAS.
 - ★ Podpora v CPU: x86-64, ARM, UltraSparc,...
 - ★ Používané v OS: MS Windows, Linux, Solaris,...
- ▶ **Segmentace:** rozdělení VAS do několika segmentů (jednorozměrných oblastí) s různými vlastnostmi a v rámci segmentů se používá stránkování.
 - ★ Podpora v CPU: POWER od IBM,...
 - ★ Používané v OS: AIX (Unix od IBM),...

Čistá segmentace

- VAS procesu je rozdělen do několika segmentů (jednorozměrných oblastí) s různými vlastnostmi implementovaných pomocí dynamických oblastí.
- Programátor/překladač může definovat vlastnosti jednotlivých segmentů.
- Pro překlad virtuálních adres se používá tabulka segmentů.
- Položka této tabulky obsahuje následující informace
 - ▶ kontrolní bity,
 - ▶ velikost segmentu (Length),
 - ▶ počáteční adresu segmentu (Base).
- Číslo segmentu (nejvýznamnější bity virtuální adresy) funguje jako index do tabulky segmentů.
- OS si musí udržovat **pro každý proces jednu tabulku segmentů**.

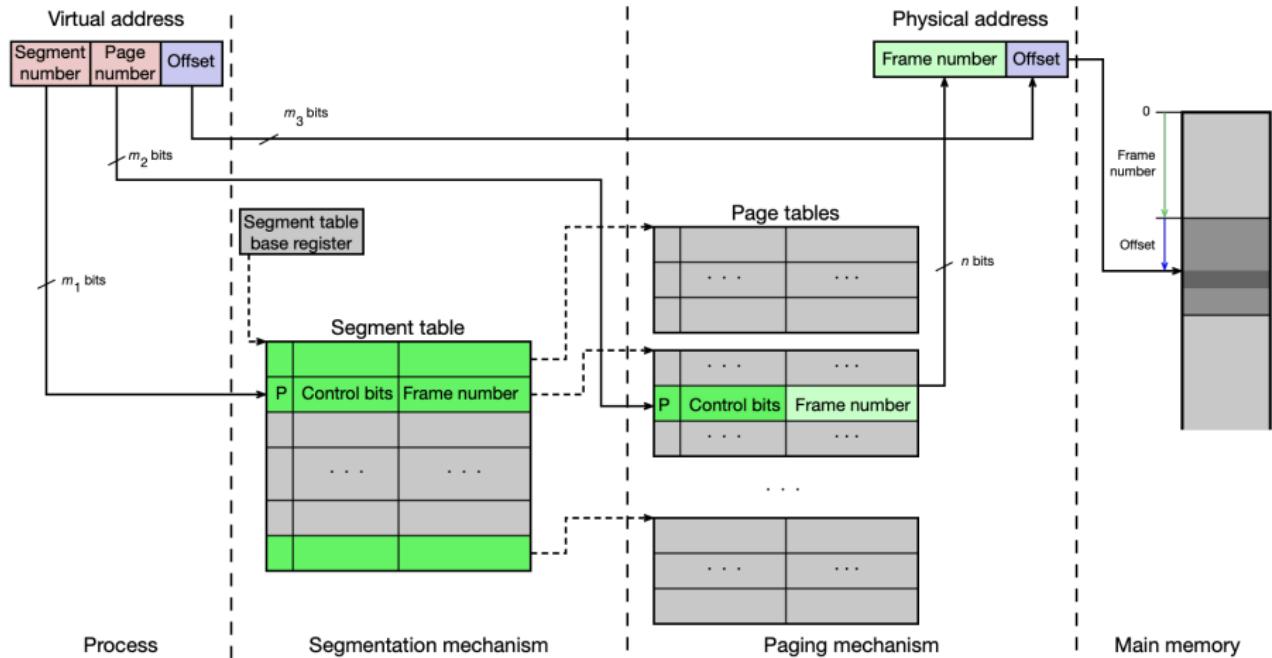
Čistá segmentace



Segmentace se stránkováním

- Při implementaci čisté segmentace vzniká podobný problém jak u dynamických oblastí
 - ⇒ problém nalezení volné oblasti pro nový segment,
 - ⇒ **segmentace se kombinuje se stránkováním a TLB.**
- **Mechanismus segmentace**
 - ▶ Číslo segmentu (nejvíce významné bity virtuální adresy) reprezentuje index do tabulky segmentů.
 - ▶ Položka v tabulce segmentů obsahuje číslo rámce, kde je uložená tabulka stránek/víceúrovňová tabulka stránek/invertovaná tabulka stránek.
- **Mechanismus stránkování**
 - ▶ Číslo stránky (méně významné bity virtuální adresy) reprezentuje index do tabulky stránek, ve které najdeme číslo rámce, kde je uložená příslušná stránka ve fyzické paměti.
 - ▶ Nejméně významné bity virtuální adresy reprezentují offset.
- **TLB**
 - ▶ Položka TLB obsahuje navíc číslo segmentu a jeho atributy.

Segmentace se stránkováním



- V okamžiku kdy většina/všechny rámce fyzické (hlavní) paměti jsou obsazené, je úkolem OS najít vhodný rámcem, jehož obsah (stránka) se uvolní. K tomu slouží algoritmy pro náhradu stránek.
- **Požadavky na algoritmy**
 - ▶ Minimalizovat počet výpadků stránek.
 - ▶ Rychlosť.
 - ▶ Jednoduchá implementace.
- **Princip algoritmů**
 - ▶ K instrukcím/datům ve VAS procesu se nepřistupuje náhodně
 - ★ Instrukce se většinou vykonávají sekvenčně, pouze občas pokračujeme instrukcí na vzdálené adresu, která je uložena na jiné stránce (větvení výpočtu, cyklus, skok,...).
 - ★ Data jsou uložena blízko sebe (halda, zásobník, pole, zřetězený seznam,...) na jedné nebo několika stránkách a přistupujeme k nim často sekvenčně (procházení pole/seznamu, push/pop na zásobník,...).
 - ⇒ Platí princip prostorové a časové lokality.

Optimální algoritmus

• Princip

- ▶ Nahradí se stránka, která má čas přístupu nejdelší (bude se k ní přistupovat za nejdelší dobu).

• Vlastnosti

- ▶ Lze dokázat, že tento algoritmus generuje minimální počet výpadků stránek.
- ▶ Nelze použít v praxi protože neznáme budoucnost
⇒ ale lze použít pro porovnání kvality reálných algoritmů.

• Příklad

- ▶ Ke stráncům se přistupovalo v pořadí: 2,3,2,1,5,2,4,5,3,2,5,2.
- ▶ Fyzická paměť se skládá ze tří prázdných rámců a, b, c.
- ▶ Jak se bude měnit obsazení rámců v průběhu času?

Číslo stránky		2	3	2	1	5	2	4	5	3	2	5	2
Rámec	a	2		2			2	4		2		2	
	b		3						3				
	c				1	5			5		5		
Výpadek stránky		x	x		x	x		x		x			

- ★ Přístup bez výpadku stránky/přístup s výpadkem stránky.
- ★ Pokud je více možností, zvolíme rámců ze začátku abecedy.
- ★ Počet výpadků: 6.

NRU algoritmus (Not Recently Used)

• Princip

- ▶ Většina systémů se stránkováním si pro každou stránku pamatuje R bit (reference) a M bit (modified).
- ▶ Při načtení stránky do paměti jsou bity nastaveny na hodnotu 0.
- ▶ Tyto bity jsou nastavovány automaticky hardwarem při každém přístupu ke stránce.
 - ★ R bit se nastaví, pokud se ke stránce přistupuje (čtení nebo zápis).
 - ★ M bit se nastaví, pokud se změnil obsah stránky (zápis).
- ▶ Abychom získali informaci, kdy se ke stránce přistupovalo (před dlouhou/krátkou dobou), je nutné, aby OS periodicky resetovat hodnotu R bitu na nulu.
- ▶ Na základě hodnot R a M bitů můžeme stránky rozdělit do čtyř tříd.
 - ★ Class 0: R=0, M=0,
 - ★ Class 1: R=0, M=1,
 - ★ Class 2: R=1, M=0,
 - ★ Class 3: R=1, M=1.
- ▶ Algoritmus NRU nahradí stránku z neprázdné třídy s nejnižším číslem.

NRU algoritmus (Not Recently Used)

• Vlastnosti

- ▶ Jednoduchý na pochopení.
- ▶ Rozumně složitá implementace.
- ▶ Relativně malý počet výpadků stránek.

NRU algoritmus (Not Recently Used)

• Příklad

- Ke stránkám se přistupovalo v pořadí: 2,3,2,1,5,2,4,5,3,2,5,2.
- Fyzická paměť se skládá z tří prázdných rámců a, b, c.
- OS resetuje R bit v časech $10 \times i$, kde $i \in \{0, 1, 2, \dots\}$.
- Jak se bude měnit obsazení rámců v průběhu času?

Číslo stránky		2	3	2		1	5		2	4	5	3		2	5	2
Typ přístupu		r	r	w		r	r		r	r	r	r		w	w	r
Čas		1	4	9	10	15	17	20	21	22	25	27	30	31	32	37
Rámce	a	Stránka	2	2				2						2	2	2
		R	1	1	0			0	1					0	1	1
		M	0	1				1						1	1	1
	b	Stránka	3			5			4		3					
		R	1	0		1	0		1		1	0				
		M	0			0			0		0					
	c	Stránka			1				5				5			
		R			1		0		1		0		0	1	1	
		M			0				0					1		
Výpadek stránky		x	x		x	x		x	x	x						

- Přístup bez výpadku stránky/přístup s výpadkem stránky.
- Reset R bitu.
- Pokud je více možností, zvolíme rámeček ze začátku abecedy.
- Počet výpadků: 7.

• Princip

- ▶ OS si udržuje **seznam všech stránek**, které se aktuálně nachází v hlavní paměti.
- ▶ V okamžiku, kdy se stránka nahraje do hlavní paměti, přidá se její záznam na konec seznamu.
- ▶ FIFO algoritmus vybere první stránku ze seznamu jako vhodného kandidáta pro nahradu.

• Vlastnosti

- ▶ Jednoduchý na pochopení a implementaci.
- ▶ Nahrazuje se stránka, které je v paměti nejdéle.
- ▶ Algoritmus nezohledňuje, kdy se ke stránce přistupovalo, ale pouze kdy se stránka nahrála do hlavní paměti
⇒ způsobuje relativně velký počet výpadků stránek.

FIFO algoritmus (First-In First-Out)

• Příklad

- ▶ Ke stránkám se přistupovalo v pořadí: 2,3,2,1,5,2,4,5,3,2,5,2.
- ▶ Fyzická paměť se skládá ze tří prázdných rámců a, b, c.
- ▶ Jak se bude měnit obsazení rámců v průběhu času?

Číslo stránky		2	3	2	1	5	2	4	5	3	2	5	2
Rámec	a	2		2		5			5	3			
	b		3			2				2	5		
	c				1		4					2	
Začátek seznamu	a	a	a	a	b	c	a	a	b	b	c	a	
Výpadek stránky	x	x	x	x	x	x	x	x	x	x	x	x	

- ★ Přístup bez výpadku stránky/přístup s výpadkem stránky.
- ★ Pokud je více možností, zvolíme rámcem ze začátku abecedy.
- ★ Počet výpadků: 9.

• Princip

- ▶ Modifikovaný FIFO algoritmus.
- ▶ Seznam stránek je implementován jako **kruhová fronta**.
- ▶ Na počátku ručička (ukazatel) ukazuje na první položku fronty.
- ▶ **Pro každou stránku si pamatujeme její R bit (reference).**
 - ★ Když se stránka nahraje do paměti, OS nastaví R bit na hodnotu 1.
 - ★ Při každém přístupu (čtení/zápis) ke stránce se nastaví R bit na hodnotu 1.
- ▶ **Postup při hledání vhodné stránky pro nahradu**
 - ★ Pokud ručka ukazuje na stránku, jejíž R bit má hodnotu 1, potom se resetuje R bit na hodnotu 0 a ručička se posune na následující stránku (položku) ve frontě.
 - ★ Předchozí krok se bude opakovat, dokud ručička nebude ukazovat na stránku s R bitem rovným hodnotě 0.
⇒ **Tato stránka se nahradí a ručička se nastaví na následující stránku ve frontě.**
- ▶ **Vlastnosti**
 - ★ Rozumně složitá implementace.
 - ★ Algoritmus generuje nízký počet výpadků stránek.

Clock algoritmus

• Příklad

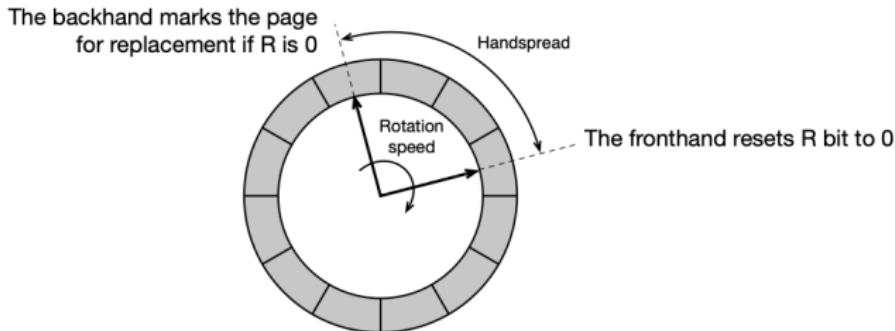
- ▶ Ke stránkám se přistupovalo v pořadí: 2,3,2,1,5,2,4,5,3,2,5,2.
- ▶ Fyzická paměť se skládá z tří prázdných rámců a, b, c.
- ▶ Jak se bude měnit obsazení rámců v průběhu času?

Číslo stránky		2	3	2	1	5	2	4	5	3	2	5	2
Rámce	a	Stránka	2	2	5			5	3				
	R	1	1	1			1	1					
	b	Stránka		3			2			2		2	
	R		1			0	1		0	1	0	1	
	c	Stránka			1		4			5			
	R			1	0	1	0		0	1			
Ručička ukazuje na		a	a	a	a	b	c	a	a	b	b	a	
Výpadek stránky		x	x		x	x	x	x	x	x	x	x	

- ★ Přístup bez výpadku stránky/přístup s výpadkem stránky.
- ★ Reset R bitu.
- ★ Pokud je více možností, zvolíme rámcem ze začátku abecedy.
- ★ Počet výpadků: 8.

Two-handed clock algoritmus

- Různé varianty Clock algoritmu jsou používány v reálných OS.
- Jako příklad může sloužit **varianta clock algoritmu se dvěma ručičkami**, který byl používán v Unixu SVR4 a v současné době ho můžeme najít v jeho nástupcích.



• Princip

- ▶ Algoritmus používá opět kruhovou frontu stránek a R bity jednotlivých stránek.
- ▶ **Obě ručičky se otáčejí stejnou rychlosťí**, která se mění podle aktuálního množství volné paměti.
- ▶ U stránky, na kterou ukazuje první ručička (fronthand), se resetuje R bit na nulu.
- ▶ Pokud stránka, na kterou ukazuje druhá ručička (backhand) má stále R bit nulový, pak je vybrána pro nahradu.
- ▶ Rozevření ručiček (handspread) společně s rychlosťí definuje časové okno, na základě kterého poznáme, zda se ke stránce nedávno přistupovalo.

LRU algoritmus (Least Recent Used)

• Princip

- ▶ Vhodným kandidátem pro náhradu je stránka, ke které se nepřistupovalo po nejdelší dobu.

• Vlastnosti

- ▶ Dobrá aproximace optimálního algoritmu.
- ▶ Problematická implementace.
 - ★ Při každém přístupu ke stránce je nutné si zapamatovat informaci o "čase" přístupu.
 - ★ Vhodným kandidátem pro náhradu je stránka s nejmenším časem (musí se porovnat "časy" všech stránek).

• Implementace pomocí speciálního hardwarového čítače

- ▶ Každá položka v tabulce stránek bude obsahovat navíc položku "time-of-used".
- ▶ Hodnota čítače bude reprezentovat logický čas a bude se inkrementovat při každém přístupu do paměti.
- ▶ Při přístupu ke stránce se uloží aktuální hodnota čítače do položky "time-of-used" v tabulce stránek.
- ▶ Vhodným kandidátem pro náhradu je stránka s nejmenší hodnotou "time-of-used".

LRU algoritmus (Least Recently Used)

• Příklad

- ▶ Ke stránkám se přistupovalo v pořadí: 2,3,2,1,5,2,4,5,3,2,5,2.
- ▶ Fyzická paměť se skládá ze tří prázdných rámců a, b, c.
- ▶ Jak se bude měnit obsazení rámců v průběhu času?

Číslo stránky		2	3	2	1	5	2	4	5	3	2	5	2
Rámec	a	2		2			2			3			
	b		3			5			5		5		
	c				1			4		2		2	
Výpadek stránky		x	x		x	x		x		x	x		

- ★ Přístup bez výpadku stránky/přístup s výpadkem stránky.
- ★ Pokud je více možností, zvolíme rámcem ze začátku abecedy.
- ★ Počet výpadků: 7.

Aging algoritmus

- Softwarová simulace LRU algoritmu.
- **Princip**
 - ▶ Pro každou stránku si systém pamatuje
 - ★ R bit (reference), který se nastaví při přístupu (čtení/zápis) ke stránce,
 - ★ n -bitový čítač C , který má všechny bity nastavené na 1 při načtení stránky do paměti.
 - ▶ Systém periodicky pro každou stránku
 - 1 posune obsah čítače C doprava o jeden bit,
 - 2 nastaví hodnotu nejvýznamejšího bitu čítače C na hodnotu R bitu,
 - 3 resetuje hodnotu R bitu na hodnotu 0.
 - ▶ Vhodným kandidátem pro nahradu bude stránka jejíž čítač C má nejmenší hodnotu.

● Vlastnosti

- ▶ Implementace tohoto algoritmu má menší režii než LRU.
- ▶ Algoritmus není tak přesný jako LRU.
 - ★ Pro každou stránku si nepamatuje přesný čas přístupu, ale pouze "interval", ve kterém se ke stránce přistupovalo.
 - ★ O každé stránce si pamatujeme pouze omezenou historii díky omezenému počtu bitů čítače C .

Aging algoritmus

• Příklad

- ▶ Systém používá pouze 6 stránek ($0, \dots, 5$), které se načetly do paměti v periodě 0, a 8-bitový čítač C .
- ▶ Zeleně orámované jsou stránky, které jsou vhodné pro nahradu na konci dané periody.

	End of Period 1	End of Period 2	End of Period 3	End of Period 4	End of Period 5
	R bits for pages 0-5				
Page	1010111	1100101	1101011	1000101	0110001
0 C:	11111111	11111111	11111111	11111111	01111111
1 C:	01111111	10111111	11011111	01101111	10110111
2 C:	11111111	01111111	00111111	00011111	10001111
3 C:	01111111	00111111	10011111	01001111	00100111
4 C:	11111111	11111111	01111111	10111111	01011111
5 C:	11111111	01111111	10111111	01011111	00101111

Aging algoritmus

• Příklad

- Ke stránkám se přistupovalo v pořadí: 2,3,2,1,5,2,4,5,3,2,5,2.
- Fyzická paměť se skládá z tří prázdných rámců a, b, c.
- OS resetuje R bit v časech $10 \times i$, kde $i \in \{0, 1, 2, \dots\}$.
- Čítač C bude reprezentován pouze 3 bity.
- Jak se bude měnit obsazení rámců v průběhu času?

Číslo stránky		2	3	2		1	5		2	4	5	3		2	5	2
Čas		1	4	9	10	15	17	20	21	22	25	27	30	31	32	37
Rámce	a	Stránka	2	2		5		4	5	3						
		R	1	1	0	1	0	1	1	1	0					
		C	7	7	7	7	7	7	7	7	7	7				
	b	Stránka	3				2						2	2		
		R	1	0			0	1					0	1	1	
		C	7	7		3	7					7	7	7		
	c	Stránka			1								5			
		R			1	0						0	1			
		C			7	7						3	7			
Výpadek stránky		x	x		x	x		x	x	x	x		x			

- Přístup bez výpadku stránky/přístup s výpadkem stránky.
- Reset R bitu.
- Pokud je více možností, zvolíme rámeček ze začátku abecedy.
- Počet výpadků: 9.

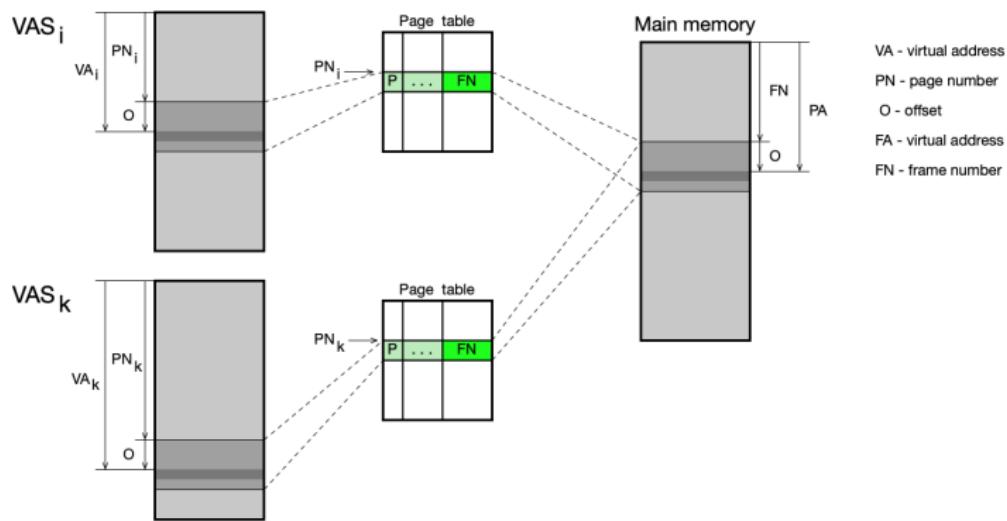
Návrh systémů se stránkováním

- Správa paměti je daleko komplexnější téma než jenom samotná architektura paměti (virtuální paměť, stránkování,...) a algoritmy pro náhradu stránek.
- Při návrhu OS je nutné vyřešit celou řadu dalších otázek, tak aby OS splňoval požadavky, které na něj klademe.
 - ▶ Jak zajistit sdílení paměti mezi více procesů?
 - ▶ Jak implementovat adresový prostor jádra OS?
 - ▶ Kdy se stránka nahraje do hlavní paměti?
 - ▶ Kde v hlavní paměti by stránka měla být umístěna?
 - ▶ Kolik fyzické paměti má být přiděleno konkrétnímu procesu?
 - ▶ Kolik procesů by mělo být maximálně ve fyzické paměti?
 - ▶ Jaký bude postup OS, když začne docházet fyzická paměť?
 - ▶ ...

Návrh systémů se stránkováním

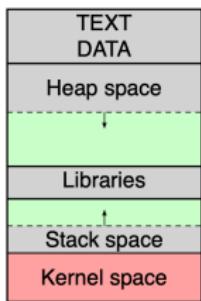
• Jak zajistit sdílení paměti mezi více procesů?

- ▶ Procesy používají soubory/struktury, které je vhodné sdílet s ostatními procesy
 - ★ spustitelné binární programy,
 - ★ knihovny,
 - ★ soubory mapované do paměti (memory mapped files),
 - ★ sdílená paměť.
- ▶ Pokud OS používá stránkování/segmentaci, pak sdílení lze implementovat relativně jednoduše.

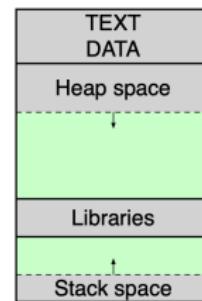


• Jak implementovat adresový prostor jádra OS?

Kernel space mapped to VAS



Kernel space separated from VAS



- ▶ Adresový prostor jádra OS mapovaný do VAS procesu
 - ★ U starších většinou 32-bitových procesorů (např. x86, SPARC V7,...) byl adresový prostor jádra mapovaný do VAS každého procesu.
 - ★ Při přechodu z "user modu" do "kernel modu" nebylo nutné měnit nastavení adresového prostoru.
- ▶ Oddělený prostor jádra
 - ★ U novějších typicky 64-bitových procesorů (např. x86-64, SPARC V9,...) je adresový prostor jádra většinou oddělený od VAS procesů.

• Kdy se stránka nahraje do hlavní paměti?

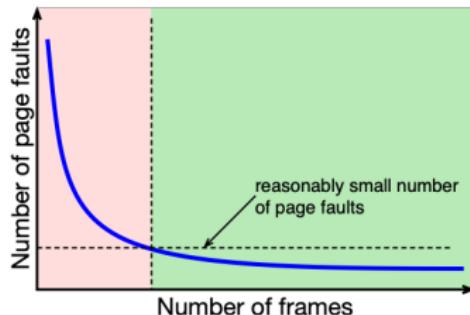
- ▶ Používají se dvě základní strategie (fetch strategies).
- ▶ Stránkování na žádost (demand paging)
 - ★ Stránka je nahrána z disku do hlavní paměti až v okamžiku, když se proces pokouší k ní přistoupit.
- ▶ Před-stránkování (prepaging)
 - ★ Tato strategie je založená na principu prostorové lokality.
 - ★ Když se proces pokusí přistoupit ke stránce, která ještě není v hlavní paměti, tak jádro načte tuto stránku a několik následujících stránek současně.
 - ⇒ minimalizuje se počet přenosů z disku,
 - ⇒ do paměti se mohou nahrát stránky, které se nebudou používat.
 - ★ Většina OS používá tuto strategii.

• Kde v hlavní paměti by stránka měla být umístěna?

- ▶ V případě používání dynamických oblastí nebo čisté segmentace se používaly různé algoritmy (např. best-fit/worst-fit,...) jejichž úkolem bylo minimalizovat fragmentaci paměti.
- ▶ Pokud systém používá stránkování, pak je nová stránka nahrána do libovolného volného rámce.

Návrh systémů se stránkováním

• Kolik fyzické paměti má být přiděleno konkrétnímu procesu?



- ▶ Graf zobrazuje závislost mezi počtem přidělených rámců fyzické paměti procesu a počtem jím generovaných výpadků stránek.
 - ★ Pokud procesu přidělíme málo rámců fyzické paměti, tak bude generovat hodně výpadků stránek (růžová oblast grafu).
 - ★ Naopak od určitého počtu přidělených rámců se počet výpadků stránek příliš nezmění i když procesu přidáme další rámce.
- ⇒ Pokud každému procesu P_i přidělíme "rozumný" počet rámců WS_i (Working Set), pak bude generovat "rozumný" počet výpadků stránek.
- ▶ **Variable-allocation strategy**
 - ★ Většina OS přiděluje rámce fyzické paměti jednotlivým procesům podle jejich aktuálních potřeb.

Návrh systémů se stránkováním

- **Kolik procesů by mělo být maximálně ve fyzické paměti?**

- ▶ V systému by měl být takový počet aktivních procesů (procesy, kde aspoň jedno vlákno je ve stavu "Ready"/"Running"), aby **součet jejich WS; byl menší než velikost hlavní paměti**
⇒ jinak budou procesy soupeřit o paměť a fyzická paměť nebude efektivně využívána.
- ▶ Úkolem administrátora je, aby toto zajistil
 - ★ omezením počtu aplikací běžících v systému,
 - ★ omezením požadavků na paměť od jednotlivých aplikací.

• Jaký bude postup OS, když začne docházet fyzická paměť?

- ▶ V hlavní paměti se můžou nacházet různá data
 - ★ jádro OS a části VAS jednotlivých procesů,
 - ★ části/celý obsah bývalých nebo aktuálně otevřených souborů,
 - ★ obsah některých pseudo FS (např. obsah adresáře /tmp),...
- ▶ OS si obvykle udržuje množinu (pool) volných rámců fyzické paměti, tak aby byl schopný okamžitě uspokojit požadavek na alokaci/načtení nové stránky.
- ▶ Pokud klesne počet volných rámců pod kritickou mez (např. definována jako parametr jádra OS), pak OS začne aktivně uvolňovat hlavní paměť.
- ▶ Pro uvolňování hlavní paměti používá OS dva mechanismy
- ▶ **Paging-out strategy**
 - ★ OS začne uvolňovat jednotlivé stránky, které byly vybrány pomocí příslušného algoritmu pro nahradu stránek.
 - ★ Modifikované stránky uloží na disk a rámce všech vybraných stránek si přidá do množiny volných rámců.
- ▶ **Swaping strategy**
 - ★ Pokud předchozí strategie nezajistí dostatečné navýšení počtu volných rámců, tak OS začne odkládat celé procesy na disk (disková oblast/soubor).
 - ★ Přednost mají procesy, kde ani jedno vlákno není ve stavu "Ready"/"Running".
 - ★ Do speciální odkládací diskové oblasti/souboru uloží pouze "anonymní" stránky, které neexistují jinde ve FS (např. halda, zásobník,...).

- ① A. S. Tanenbaum, H. Bos: *Modern Operating Systems (4th edition)*, Pearson, 2014.
- ② W. Stallings: *Operating Systems: Internals and Design Principles (9th edition)*, Pearson, 2017.
- ③ A. Silberschatz, P. B. Galvin, G. Gagne: *Operating System Concepts (9th edition)*, Wiley, 2012.
- ④ R. McDougall, J. Mauro: *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture (2nd edition)*, Prentice Hall, 2006.

Operační systémy

Datová úložiště

Jan Trdlička



České vysoké učení technické v Praze, Fakulta informačních technologií
Katedra počítačových systémů

<https://courses.fit.cvut.cz/BI-OSY>

Obsah přednášky

- 1 Datové uložiště
- 2 HDD (Hard Disk Drive)
- 3 SSD (Solid State Drive)
- 4 RAID (Redundant Array of Independent Disks)
 - RAID 0 (concatenation, striping)
 - RAID 1 (mirroring)
 - RAID 1 + 0 (mirroring + striping)
 - RAID 2, 3, 4
 - RAID 5
 - RAID 6
- 5 Typy připojení datového úložiště k výpočetnímu systému
 - DAS (Direct-Attached Storage)
 - NAS (Network-Attached Storage)
 - SAN (Storage Area Network)

Datové uložiště

- Hardware, který slouží k dlouhodobému uložení informací.
- Někdy je také označován jako "sekundární paměť" (secondary storage) nebo externí paměť (external memory).
- Úložný prostor, který datové uložiště nabízí,
 - ▶ se skládá ze sektorů (nejménší adresovatelná jednotka, 512B/4KB).
 - ▶ je obvykle rozdělen na menší oblasti (např. diskové oblasti v případě disků nebo "volumes" v případě RAID), které jsou spravovány prostřednictvím
 - ★ OS: oblast obsahuje systém souborů (FS), data jsou zde uložena ve formě souborů, OS poskytuje aplikacím rozhraní na úrovni FS.
 - ★ aplikace: data jsou zde uložena v proprietárním formátu definován konkrétní aplikací (např. databáze), OS poskytuje pouze rozhraní na úrovni sektorů.
- V současné době mezi nejběžnější typy datových úložišť patří
 - ▶ HDD (Hard Disk Drive) neboli pevný disk,
 - ▶ SSD (Solid State Drive),
 - ▶ RAID (Redundant Array of Independent Disks).
- Datové úložiště může být připojeno k systému několika způsoby
 - ▶ DAS (Direct Attached Storage),
 - ▶ NAS (Network Arrea Storage),
 - ▶ SAN (Storage Arrea Network).

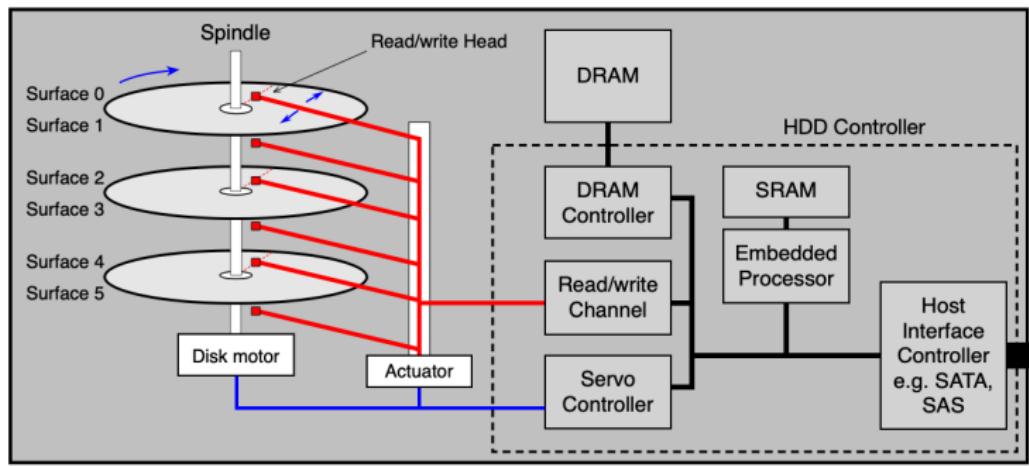
HDD (Hard Disk Drive): Architektura

• Mechanické části

- ▶ Několik ploten, které mají dva povrchy a otáčí stejnou rychlosí.
- ▶ Pohyblivé hlavičky, které umožňují čtení/zápis z příslušného povrchu a nacházejí se všechny vždy ve stejné vzdálenosti od středu povrchu.

• Elektrické části

- ▶ Řadič disku: obsahuje procesor, paměť s firmwarem a příslušné řadiče.
- ▶ Vyrovnávací paměť (DRAM): obsahuje čtená/zapisovaná data a může mít velikost několik stovek MB.



HDD: Geometrie

● Sektor (Sector)

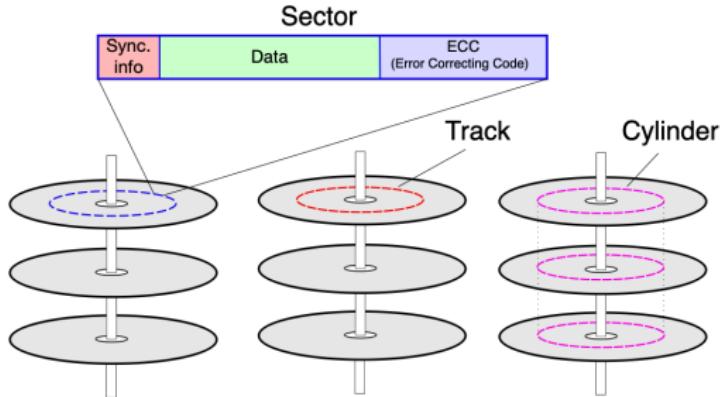
- ▶ Nejmenší adresovatelná jednotka na disku, která obsahuje
 - ★ synchronizační informace sloužící pro řadič disku,
 - ★ data ("nově" 4096 B, dříve 512 B),
 - ★ ECC (Error Correction Code) pro detekci a opravu chyb.

● Stopa (Track)

- ▶ Množina sektorů na jednom povrchu ve stejné vzdálenosti od středu.
- ▶ Počet sektorů ve stopě se může lišit v závislosti na poloměru.
- ▶ Stopy se číslují od vnějšího okraje povrchu.

● Cylindr (Cylinder)

- ▶ Množina všech stop o daném poloměru na všech površích.



● Adresování sektorů

- ▶ CHS (Cylinder Head Sector)
 - ★ Starší adresování podle geometrie disku.
 - ★ Například sektor [1, 2, 3] představuje data na 1. cylindru, 2. povrchu a ve 3. sektoru.
- ▶ LBA (Logical Block Addressing)
 - ★ "Novější" adresování.
 - ★ Sektory jsou číslovány sekvenčně od nuly, začíná se od vnějšího okraje cylindru: [0, 0, 0] → 0, [0, 1, 0] → 1, [0, 2, 0] → 2, ...

● Zone Bit Recording (ZBR)

- ▶ Stopy rozděleny do zón.
- ▶ V rámci zóny je konstantní počet sektorů na stopu.
- ▶ Vnější zóny mají větší počet sektorů na stopu než zóny blíže ke středu.

HDD: Host Interface Controller

- HDD může být připojen k systému pomocí různých typů sběrnic s různými vlastnostmi.
- **Menší systémy (osobní počítače)**
 - ▶ SATA (Serial ATA)
 - ★ Sériová sběrnice, rychlosti: 1.5 Gb/s, 3 Gb/s, 6 Gb/s,
 - ★ vzdálenost: 1 m.
 - ▶ Thunderbolt
 - ★ Seriová sběrnice spojující PCI-Express a Display port,
 - ★ rychlosti: 40 Gb/s.
- **"Enterprise" systémy (velké dražší servery)**
 - ▶ SAS (Serial Attached SCSI)
 - ★ Sériová sběrnice, rychlosti: 22.5 Gb/s, vzdálenost: 10m.
 - ▶ FC (Fibre Channel)
 - ★ Sériová sběrnice, rychlosti: 128 Gb/s
 - ★ vzdálenosti: 10km.

HDD: Rychlosť prístupu k datám

- **Na čom závisí rychlosť čtenia/sápisu jednoho sektoru?**

- ▶ Doba vystavenia (seek time)
 - ★ Čas nastavenia hlavičiek nad správny cylindr (cca 1-10ms).
- ▶ Průměrné rotační zpoždění (rotational delay)
 - ★ Čas posunutia správneho sektoru pod hlavičku,
 - ★ Při rotaci disku 5 000-15 000 rpm (rotations per minute)
⇒ průměrné rotační zpoždění je 6-2ms.
- ▶ Čas prenosu dat.

- **OS/aplikace je odpovědný/á za efektivní používání disku.**

- ▶ minimalizování doby vystavení,
- ▶ maximalizace počtu přenesených bytů za čas.

- **Algoritmy plánování prístupu na disk**

- ▶ Dříve implementované pouze v OS, nyní v řadičích disků.
- ▶ Určují pořadí zpracování jednotlivých požadavků.
 - ★ Tagged Command Queuing (TCQ) pro SCSI a ATA disky,
 - ★ Native Command Queuing (NCQ) pro SATA.

HDD: Příklad sekvenční x náhodný přístup k datům

- **Mějme disk s těmito parametry**

- ▶ HDD má pouze jeden povrch, který se otáčí rychlosťí 10000 rpm.
- ▶ Velikost sektoru 512 B.
- ▶ Každá stopa má 320 sektorů.
- ▶ Průměrný čas vystavení hlaviček (seek time) je 10 ms.
- ▶ Vystavení hlaviček nad sousední stopu (track-to-track seek time) je 1ms.

- **Jaké je průměrné rotační zpoždění?**

- ▶ $\frac{0+(60/10000)}{2} = 3 \text{ ms}$

- **Kolik stop potřebuji na uložení 2560 sektorů dat?**

- ▶ $2560/320 = 8 \text{ stop}$

- **Jak dlouho bude trvat přečíst 2560 sektorů uložených na sousedních stopách?**

- ▶ Načtení sektorů na první stopě: $10 + 3 + 6 = 19 \text{ ms}$.
- ▶ Načtení sektorů na následující sousední stopě: $1 + 3 + 6 = 10 \text{ ms}$.
- ▶ Celková doba čtení všech sektorů: $19 + 7 \times 10 = 89 \text{ ms}$.

- **Jak dlouho bude trvat přečíst 2560 sektorů uložených náhodně na disku?**

- ▶ Načtení jedno sektoru: $10 + 3 + 6/320 = 13.01875 \text{ ms}$.

- ▶ Celková doba čtení všech sektorů: $2560 \times 13.01875 = 33.328 \text{ s.}$

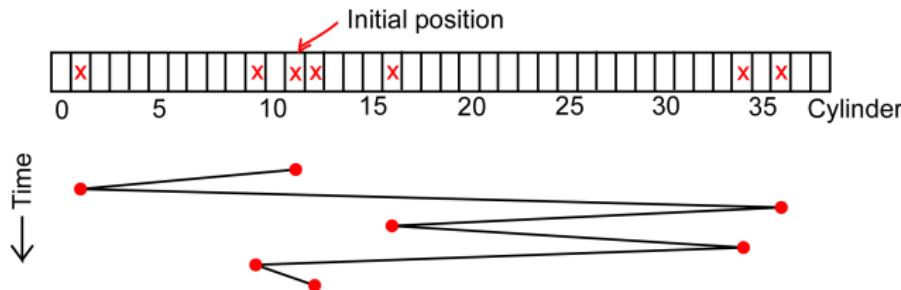


- **First-In-First-OUT (FIFO)**

- ▶ Požadavky (čtení/zápis) jsou řazeny do fronty.
- ▶ Požadavky budou obslouženy v pořadí v jakém přišly.
- ▶ **Výhody:** spravedlnost.
- ▶ **Nevýhody:** horší výkon.

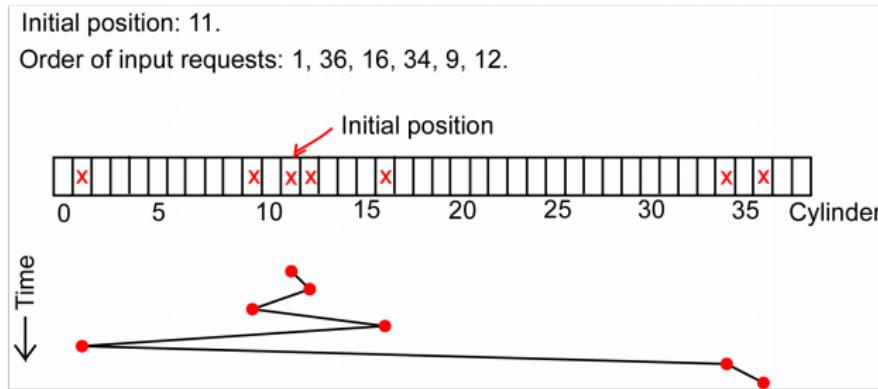
Initial position: 11.

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



• Shortest Service Time First (SSTF)

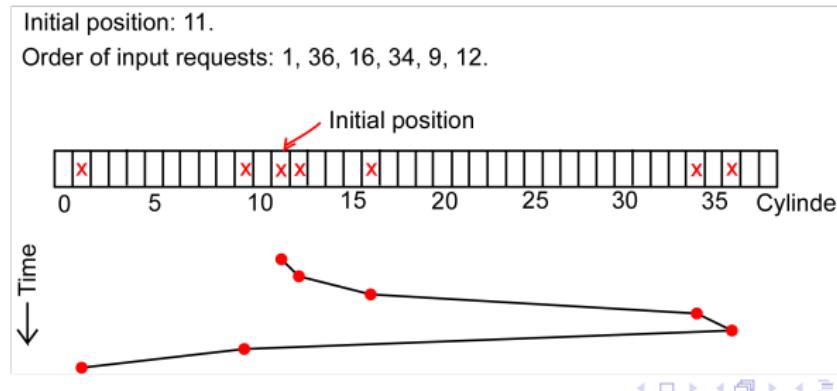
- ▶ Požadavky (čtení/zápis) jsou řazeny do fronty.
- ▶ Nejdříve jsou obslouženy požadavky, které vyžadují nejmenší pohyb hlaviček z aktuální pozice.
- ▶ **Výhody:** lepší výkon než u FIFO.
- ▶ **Nevýhody:**
 - ★ Hlavičky mají tendenci setrvávat uprostřed disku.
 - ★ Vzniká problém stárnutí u požadavků z krajních pozic.



HDD: Algoritmy plánování přístupu na disk

● SCAN Algorithm (elevator alg.)

- ▶ Požadavky (čtení/zápis) jsou řazeny do fronty.
- ▶ 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 uspokojí se všechny požadavky v druhém směru. Toto postup se opakuje.
- ▶ **Výhody:** částečně se omezil problém stárnutí požadavků.
- ▶ **Nevýhody**
 - ★ Trochu horší výkon než SSTF algoritmus.
 - ★ Neřeší stárnutí při velkém počtu požadavků v úzké oblasti cylindrů.

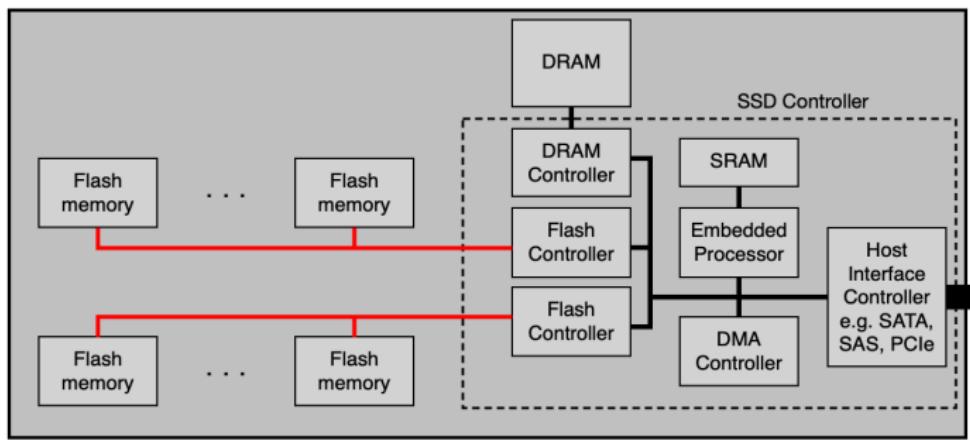


• ***N-step SCAN***

- ▶ Vylepšená verze SCAN algoritmu, který odstraňuje problém, strárnutí požadavků.
- ▶ Původní fronta požadavků je rozdělena na několik front délky N , které se postupně plní požadavky.
- ▶ Jednotlivé fronty jsou zpracovány postupně. Požadavky z jedné fronty jsou obslouženy pomocí SCAN algoritmu.
- ▶ Tento algoritmus je zobecněním předchozích algoritmů.
 - ★ Pokud bude $N = 1$, pak se bude chovat jako FIFO algoritmus.
 - ★ Pokud bude $N \rightarrow \infty$, pak se bude chovat jako SCAN algoritmus.
- ▶ **Výhody:**
 - ★ Omezil se problém strárnutí požadavků, protože je garantováno, že požadavek může být předbehnut maximálně $N - 1$ jinými požadavky.
- ▶ **Nevýhoda:** trochu horší výkon než SCAN algoritmus.

SSD (Solid State Drive): Architektura

- Neobsahuje žádné "mechanické" části jako HDD
⇒ přístup k datům, který není závislý na umístění dat.
- Výhody
 - ▶ Rychlý přístup k datům.
 - ▶ Menší spotřeba, menší rozměry.
- Nevýhody
 - ▶ Menší kapacita.
 - ▶ Vyšší cena za byte.



Porovnání HDD a SSD

Parametr	HDD [5]	SSD [6]	
I/O interface	SATA/SAS	SATA/SAS	NVMe
Sequential Read (MB/s)	250	555	6 800
Sequential Write (MB/s)	250	520	6 000
Random Read (4KB) IOPS	240	98 000	1 000 000
Random Write (4KB) IOPS	240	75 000	180 000
Latency Read (ms)	Average 4.16	Max 0.4	0.1
Latency Write (ms)	Average 4.16	Max 4.0	0.03
Capacity	14 TB	3.84 TB	3.2 TB

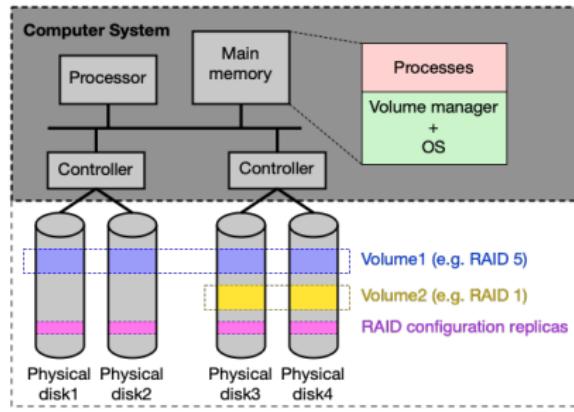
- Pokud potřebujeme
 - ▶ větší kapacitu datového úložiště,
 - ▶ větší rychlosť sekvenčního čtení/zápisu,
 - ▶ větší počet R/W operací za sekundu (IOPS),
 - ▶ větší spolehlivost (dostupnost dat při výpadku mechaniky disku/flash paměti, diskového řadiče, připojení,...),
⇒ pak je řešením RAID.

RAID (Redundant Array of Independent Disks)

- Myšlenka publikována v 1988 na univerzitě California Berkeley.
- Datové úložiště se skládá z množiny fyzických disků (v ideálním případě stejných disků HDD/SSD) a data jsou na ně ukládána (mapována) různými způsoby (různé typy RAID: 0, 1, 1+0, 5, 6, ...) a tím je dosahováno různých vlastností datového úložiště.
- Kromě RAID 0 jsou všechny ostatní typy redundantní ⇒ část kapacity datového úložiště obsahuje redundantní informace, což umožňuje, že data jsou dostupná i v případě výpadku jednoho nebo více fyzických disků.
- Výhodou RAID oproti HDD/SSD je, že většina jeho vlastností lze efektivně konfigurovat prostřednictvím typu RAID a jeho parametrů.
 - ▶ Kapacita.
 - ▶ Rychlosť sekvenčního čtení/zápisu.
 - ▶ Počet R/W operací za sekundu.
 - ▶ Spolehlivost.
- RAID je v praxi implementován dvěma základním způsoby
 - ▶ softwarový RAID,
 - ▶ hardwarový RAID.

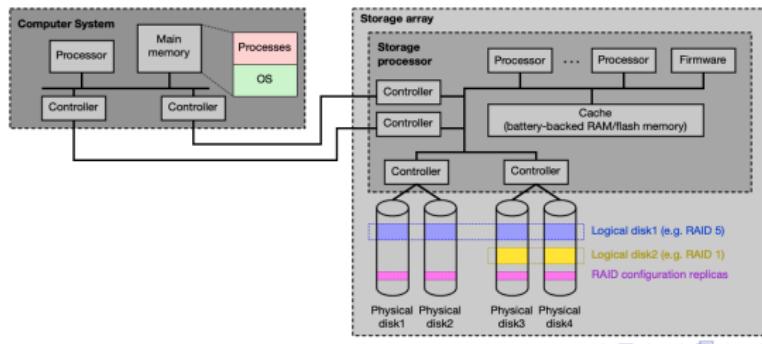
Softwarový RAID

- Fyzické disky HDD/SSD (disk1,..., disk4) jsou k systému připojeny přes příslušné sběrnice a jsou standardně spravované prostřednictvím OS .
- **Volume manager (VM)**
 - ▶ Software, který **ukládá (mapuje)** data na jednotlivé fyzické disky a provádí nutné **výpočty** (např. výpočet parity,...).
 - ▶ Spravuje logické disky (volumes), které představují konkrétní typ RAIDu a **poskytuje interface, přes který k nim může přistupovat OS a jednotlivé aplikace.**
 - ▶ Konfigurace celého VM je uložena na fyzických discích.
 - ▶ Příklady reálných VM
 - ★ Logical VM (Linux), Solaris VM (Solaris),...
 - ★ Veritas VM (MS Windows, Linux, Solaris, AIX,...).



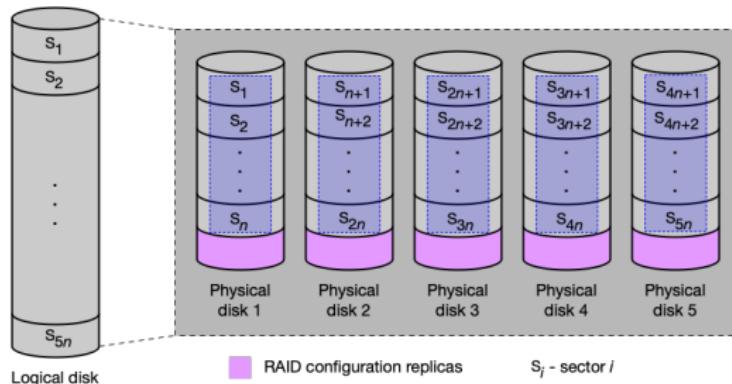
Hardwareový RAID

- Reprezentován speciálním hardwarem, který obsahuje
 - ▶ jeden nebo několik procesorů,
 - ▶ skrytou paměť, která je chráněna proti výpadku napájení a slouží pro dočasné uložení dat,
 - ▶ paměť s firmwarem,
 - ▶ fyzické disky (HDD/SSD), které jsou připojeny příslušnými sběrnicemi k systému.
- Firmware
 - ▶ Stará se o ukládání (mapování) dat na jednotlivé fyzické disky a provádí příslušné výpočty (např. výpočet parity,...).
 - ▶ Spravuje logické disky, které představují konkrétní typ RAIDu a poskytuje k nim interface pro OS/aplikace, které běží na připojeném výpočetním systému (OS přímo nevidí fyzické disky HW RAIDu).
 - ▶ Poskytuje interface pro konfigurování a monitorování HW RAIDu.



RAID 0 – zřetězení (concatenation)

- Někdy také označováván jako JBOD (Just a Bunch Of Disks).
- **Princip**
 - ▶ Data jsou ukládána/mapována postupně na jednotlivé fyzické disky (jakmile se zaplní první disk, data se začnou ukládat na druhý disk, ...).
- **Vlastnosti**
 - ▶ Redundance je 0%
⇒ výpadek jednoho disku způsobí ztrátu všech dat.
 - ▶ Výkon logického disku je skoro stejný jako výkon jednoho fyzického disku.
- **Použití**
 - ▶ Navýšení kapacity diskového úložiště.



RAID 0 – prokládání (striping)

● Princip

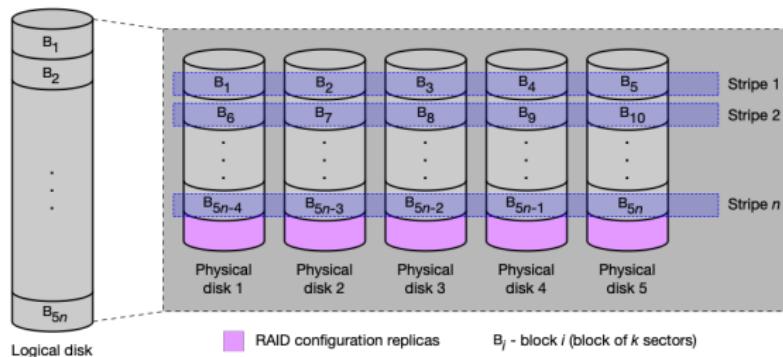
- ▶ Při vytváření RAIDu administrátor definuje blok jako k sousedních sektorů.
- ▶ Data jsou ukládána/mapována cyklicky po blocích na jednotlivé fyzické disky (jakmile se zaplní první "stripe", data se začnou ukládat na druhý "stripe", ...).

● Vlastnosti

- ▶ Redundance je 0% ⇒ výpadek jednoho disku způsobí ztrátu všech dat.
- ▶ Nechť m je počet fyzických disků.
- ▶ R/W operace se zrychlí až m krát, pokud velikost dat bude m bloků.
- ▶ Počet R/W operací za sekundu se zvýší až m krát, pokud velikost dat bude jeden blok.

● Použití

- ▶ Navýšení kapacity a výkonu diskového úložiště.



RAID 1 – zrcadlení (mirroring)

● Princip

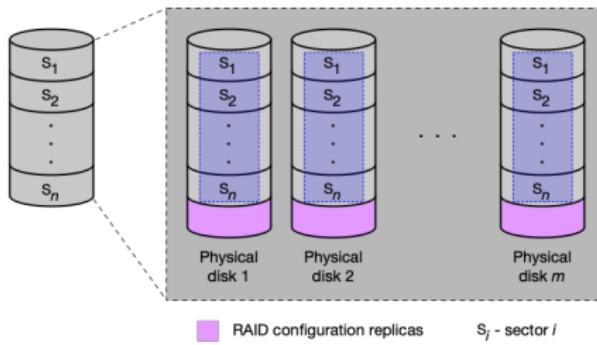
- ▶ Stejná data jsou ukládána/mapována na všechny fyzické disky (každý disk obsahuje stejnou kopii dat). RAID 1 běžně obsahuje dvě kopie dat/dva fyzické disky, ale obecně může obsahovat m kopií dat/fyzických disků.

● Vlastnosti

- ▶ Redundance je $100 \times (m - 1)/m \%$
⇒ data přežijí výpadek $m - 1$ disků a výkon nebude degradován.
- ▶ R/W operace bude přibližně stejně rychlá jako u fyzického disku.
- ▶ Počet Read operací za sekundu se zvýší až m krát a počet Write operací za sekundu bude přibližně stejný jako u fyzického disku.

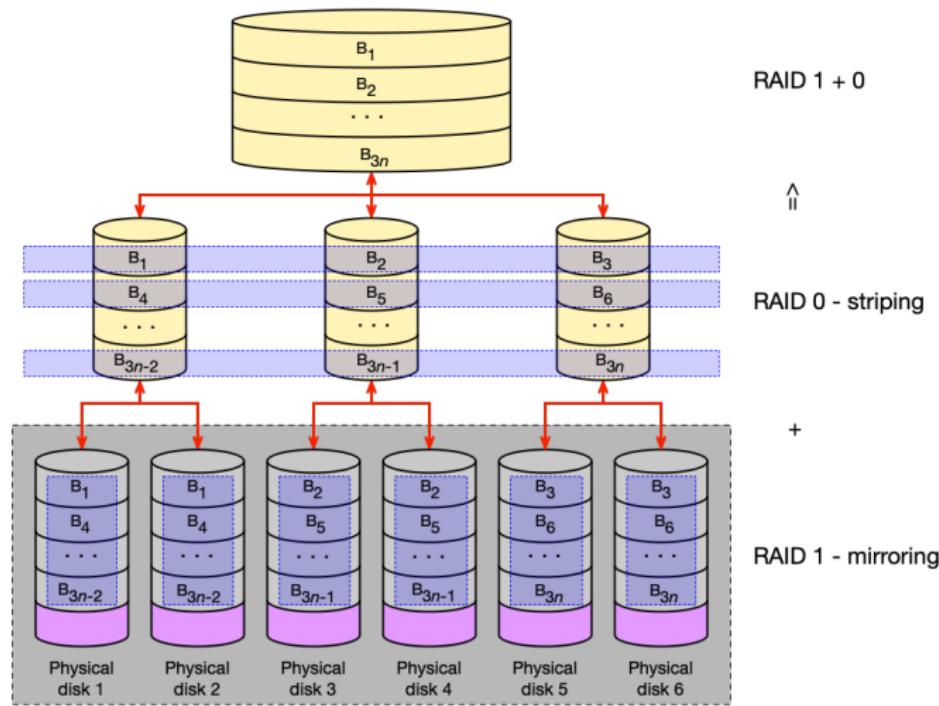
● Použití

- ▶ Zabezpečení dat na datovém úložišti.



RAID 1+0 (stripe), RAID 10

- RAID 1+0 je kombinací RAIDu 1 (zrcadlení) a RAIDu 0 (prokládání), tak aby výsledný RAID získal dobré vlastnosti z obou typů RAIDů.



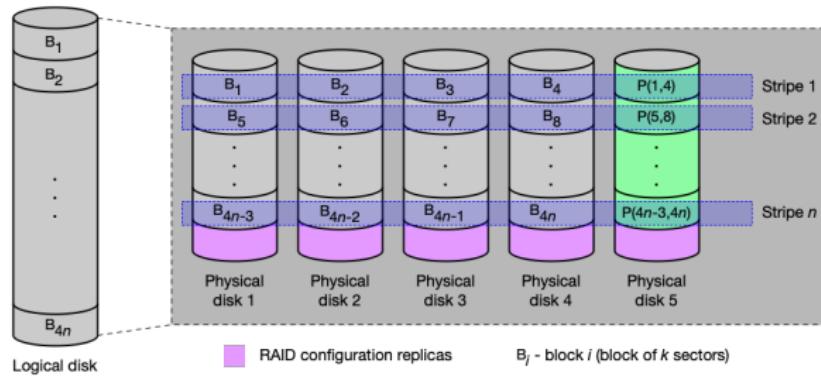
RAID 1+0 (stripe), RAID 10

● Vlastnosti

- ▶ Redundance je 50 % (pokud zrcadlení bude mít pouze dvě kopie dat)
⇒ data přežijí teoreticky výpadek $m/2$ disků a výkon nebude degradován.
- ▶ R/W operace se zrychlí až $(m/2)$ krát, pokud velikost dat bude $m/2$ bloků.
- ▶ Počet Read operací za sekundu se zvýší až m krát, pokud velikost dat bude jeden blok.
- ▶ Počet Write operací za sekundu se zvýší až $(m/2)$ krát, pokud velikost dat bude jeden blok.
- ▶ Při výpadku jednoho fyzického disku bude doba obnovy úměrná kapacitě jednoho fyzického disku.

Raid 2, 3, 4

- RAID 2, 3 a 4 se běžně v datových úložištích nepoužívají.
- **RAID 2**
 - ▶ Prokládání po bitech + zabezpečení pomocí Hammingova kódu.
- **RAID 3**
 - ▶ Prokládání po bytech + zabezpečení pomocí parity uložené na jednom fyzickém disku.
- **RAID 4**
 - ▶ Prokládání po blocích + zabezpečení pomocí parity uložené na jednom fyzickém disku. Parita je definována jako $P(i, j) = B_i \text{ XOR } B_{i+1} \text{ XOR } \dots \text{ XOR } B_j$.
 - ▶ Problém: paritní disk může být při větším počtu zápisů přetížen.



Raid 5 – prokládání s distribuovanou paritou

• Princip

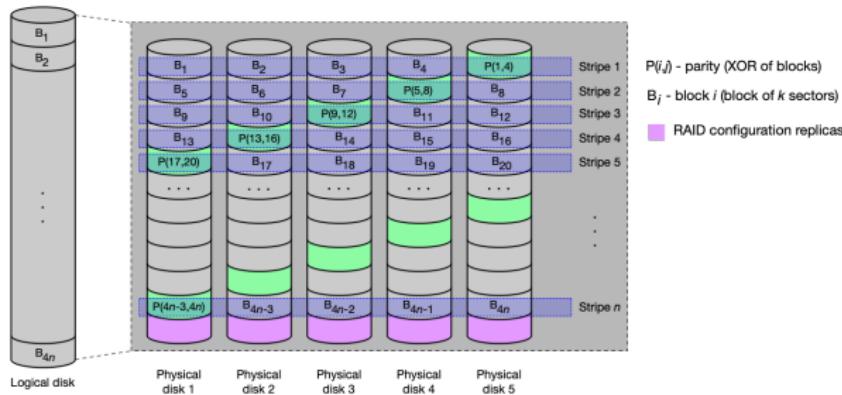
- ▶ Prokládání po blocích na m fyzických discích + zabezpečení pomocí parity cyklicky ukládané na jednotlivých fyzických discích.

• Vlastnosti

- ▶ Redundance je $100/m\%$ ⇒ při výpadku jednoho disku budou data ještě dostupná, ale bude degradován výkon.
- ▶ Read operace se zrychlí až $(m - 1)$ krát, pokud velikost dat bude $(m - 1)$ bloků.
- ▶ Write operace pomalejší vzhledem k SW RAIDu!
- ▶ Počet R operací za sekundu se zvýší až m krát, pokud velikost dat bude jeden blok.

• Použití

- ▶ Navýšení kapacity, zabezpečení dat a navýšení výkonu u Read operací.



Raid 6 – prokládání s distribuovanou paritou

• Princip

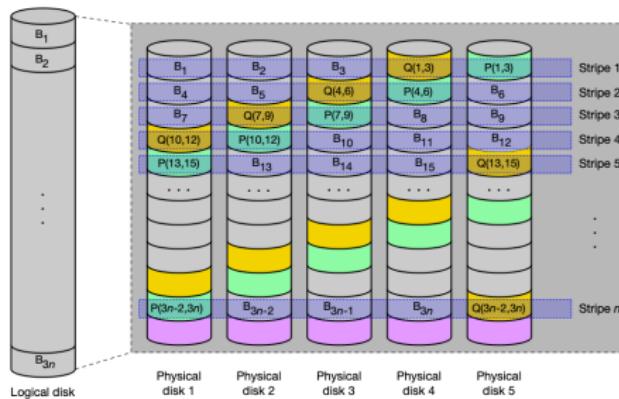
- ▶ Prokládání po blocích na m fyzických discích + zabezpečení pomocí dvojí parity cyklicky ukládané na jednotlivých fyzických discích.

• Vlastnosti

- ▶ Redundance je $200/m\%$ ⇒ při výpadku dvou disků budou data ještě dostupná, ale bude degradován výkon.
- ▶ Read operace se zrychlí až $(m - 2)$ krát, pokud velikost dat bude $(m - 1)$ bloků.
- ▶ Write operace pomalejší vzhledem k SW RAIDu!
- ▶ Počet R operací za sekundu se zvýší až m krát, pokud velikost dat bude jeden blok.

• Použití

- ▶ Navýšení kapacity, zabezpečení dat a navýšení výkonu u Read operací.



$P(i,j)$ - parity (XOR of blocks)

$Q(i,j)$ - different coding can be used

- 1) Reed-Solomon coding
- 2) EvenOdd coding
- 3) RDP coding
- 4) Liberation coding, ...

B_j - block i (block of k sectors)

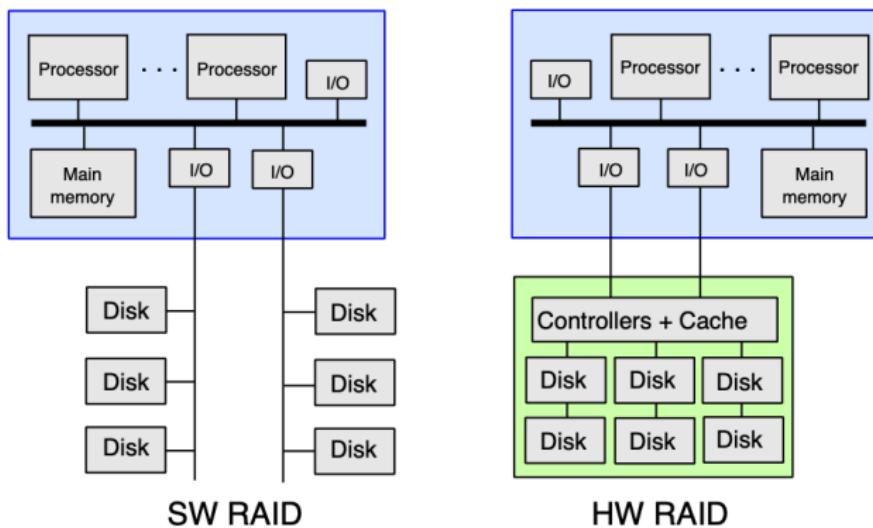
Pink square - RAID configuration replicas

Porovnání různých typů RAID

Vlastnosti	RAID 0 striping	RAID 1	RAID 10	RAID 5	RAID 6
Minimální počet disků	2	2	4	3	4
Ochrana dat	Žádná ochrana	Výpadek jednoho disku	Výpadek jednoho disku	Výpadek jednoho disku	Výpadek dvou disků
Výkon čtení	Vysoký	Vysoký	Vysoký	Vysoký	Vysoký
Výkon zápisu	Vysoký	Střední	Střední	Nízký	Nízký
Výkon čtení (při výpadku disku)	–	Střední	Vysoký	Nízký	Nízký
Výkon zápisu (při výpadku)	–	Vysoký	Vysoký	Nízký	Nízký
Využitá kapacita	100%	50%	50%	67%-94%	50%-88%

DAS (Direct-Attached Storage)

- Úložiště (HDD/SSD/RAID) je k systému připojeno přímo přes V/V porty systému.
- OS/aplikace vidí jednotlivé sektory v úložišti.
- **Technologie připojení**
 - ▶ SATA, SAS, Ethernet, Fibre channel,...
- **Vlastnosti**
 - ▶ Některé technologie (např. SCSI) umožňují "multihosting" (jedno úložiště lze připojit k více systémům současně).



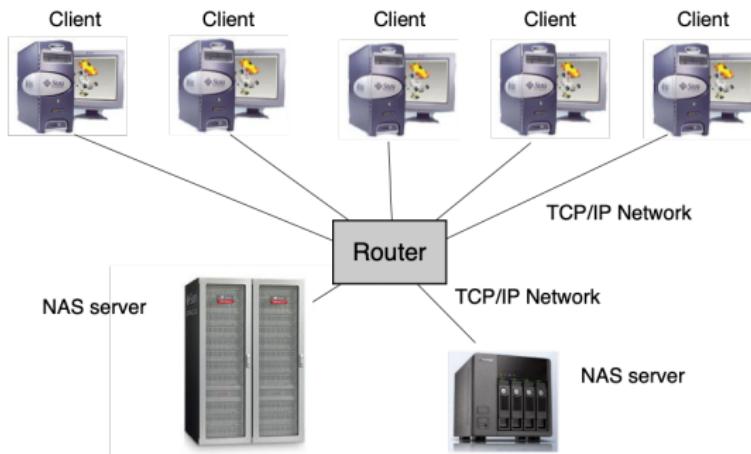
NAS (Network-Attached Storage)

- **NAS server**

- ▶ Výpočetní systém + OS.
- ▶ Datové úložiště (HDD/SSD/RAID).

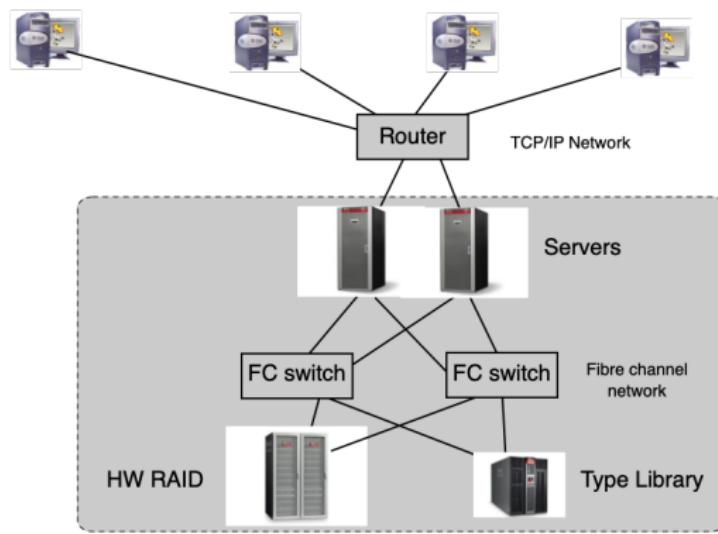
- **Technologie připojení**

- ▶ Data jsou přístupná na úrovni distribuovaného systému souborů přes příslušné protokoly
 - ★ NFS (Network File System),
 - ★ SMB (Server Message Block), Samba, CIFS (Common Internet File System).



SAN (Storage Area Network)

- Datová úložiště jsou na oddělené síti.
- OS/aplikace vidí jednotlivé sektory v úložišti.
- **Technologie připojení**
 - ▶ Ethernet, Fibre channel,...
- **Vlastnosti**
 - ▶ Multihosting (jedno úložiště lze připojit k více systémům současně).
 - ▶ Multipathing (mezi datovým úložištěm a systémem existuje více nezávislých cest).



Použité zdroje

- ① A. S. Tanenbaum, H. Bos: *Modern Operating Systems (4th edition)*, Pearson, 2014.
- ② W. Stallings: *Operating Systems: Internals and Design Principles (9th edition)*, Pearson, 2017.
- ③ A. Silberschatz, P. B. Galvin, G. Gagne: *Operating System Concepts (9th edition)*, Wiley, 2012.
- ④ R. McDougall, J. Mauro: *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture (2nd edition)*, Prentice Hall, 2006.
- ⑤ *Seagate HDD*. [Online]. Available:
https://www.seagate.com/www-content/datasheets/pdfs/ironwolf-pro-14tb-DS1914-7-1807US-en_US.pdf. [Accessed: 23-Apr-2019].
- ⑥ *Kingston SSD*. [Online]. Available:
<https://www.kingston.com/en/ssd/enterprise/DC500R>. [Accessed: 23-Apr-2019].

Operační systémy

Systémy souborů I

Jan Trdlička



České vysoké učení technické v Praze, Fakulta informačních technologií
Katedra počítačových systémů

<https://courses.fit.cvut.cz/BI-OSY>

Obsah přednášky

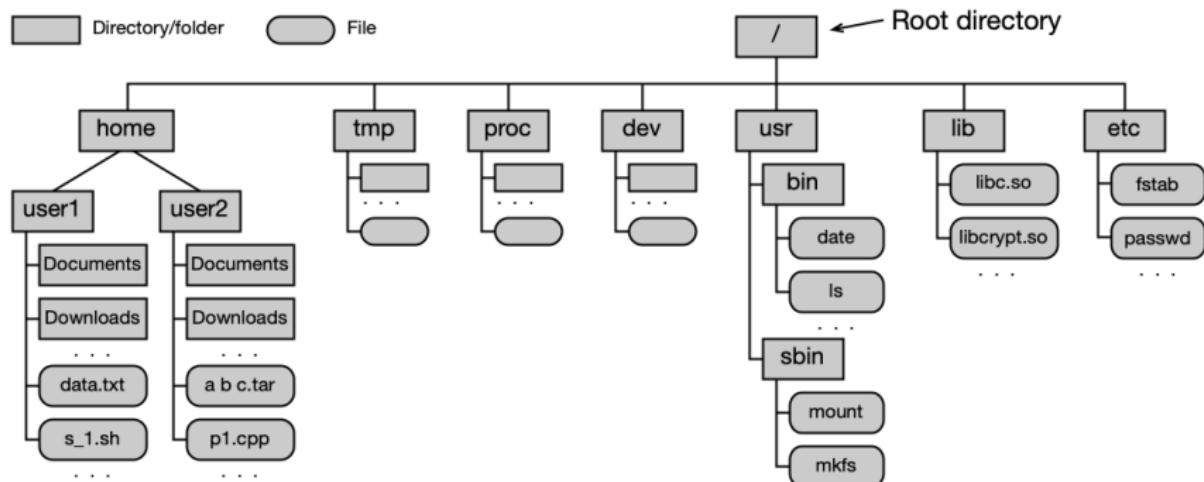
- 1 FS z pohledu uživatele
- 2 FS z pohledu administrátora
- 3 Implementace FS
 - Rozložení dat ve FS
 - Informace o volných datových strukturách
 - Alokace obsahu souboru
 - po souvislých oblastech datových bloků
 - po jednotlivých datových blocích (FAT/i-node)
 - Implementace adresářů
 - Sdílené soubory
 - Příklady: FAT, UFS

FS z pohledu uživatele

● Uživatel

- ▶ Data uložená v různých systémech souborů vidí jako jeden (Unix) nebo několik stromů adresářů (MS Windows), které pro něj představují homogenní strukturu.
- ▶ K těmto datům přistupuje prostřednictvím stejných aplikací (GUI), příkazů (CLI) nebo systémových volání/funkcí (API), bez ohledu na to, jestli data jsou uložena na lokálním, pseudo nebo vzdáleném FS.

● Příklad stromu adresářů v OS unixového typu



FS z pohledu uživatele

● Adresáře/Složky

- ▶ umožňují hierarchické uspořádání dat do stromové struktury podle požadované logiky.
- ▶ Pozici adresáře/souboru ve stromu definujeme cestou, kterou můžeme vyjádřit dvěma způsoby
 - ★ **absolutní cesta:** vztázená ke kořeni stromu (kořenový adresář),
 - ★ **relativní cesta:** vztázená k aktuální pozici ve stromu (pracovní adresář).
- ▶ Jako oddělovač adresářů v cestě se používají v různých OS různé znaky
 - ★ MS Windows: C:\Users\User1\Documents\data.txt
 - ★ OS unixového typu: /home/user1/data.txt

● Typické operace nad adresářem

- ▶ **Create ()**: vytvoří nový adresář (v Unixu adresář obsahuje podadresáře . a ..).
- ▶ **Delete ()**: smaže adresář.
- ▶ **Opendir ()**: do hlavní paměti se z FS načtou potřebné informace o adresáři (atributy, diskové adresy,...).
- ▶ **Closedir ()**: uvolní se příslušné datové struktury v hlavní paměti, které související s adresářem.
- ▶ **Readdir ()**: načte se následující položka adresáře.
- ▶ **Seekdir ()**: nastaví pozici následující položky pro Readdir () .
- ▶ **Rename ()**: přejmenuje adresář.
- ▶ **Link ()**: vytvoří link na adresář.
- ▶ **Unlink ()**: smaže link na adresář.

FS z pohledu uživatele

● Soubor

- ▶ Slouží k uložení informace/dat ve FS a k jejímu pozdějšímu použití.
- ▶ Soubor je ve FS reprezentován: jménem, atributy a obsahem.

▶ Jméno souboru

- ★ Jméno souboru společně s cestou slouží k určení pozice ve stromě adresářů.
- ★ V různých FS můžou být různé požadavky na jméno (typ kódování, rozlišování malých/velkých písmen, používání speciálních znaků, maximální délka,...).

▶ Atributy souboru

- ★ Definují vlastnosti souboru a patří mezi ně následující atributy.
- ★ Typ: obyčejný soubor, adresář, link,...
- ★ Vlastníci souboru: uživatel, skupina, ostatní,...
- ★ Přístupová práva: pro čtení, modifikaci a spuštění, setuid-bit, ACL práva,...
- ★ Různé časy: čas přístupu, modifika,...

▶ Obsah souboru

- ★ Představuje samotnou informaci/data, která jsou uložena v datových blocích ve FS.
- ★ Většina OS (MS Windows, Unix,...) vidí obsah souboru pouze jako pole bytů a jeho interpretace je na jednotlivých procesech (vyjímkou jsou spuštěné soubory).

- **Typické operace nad souborem**

- ▶ **Create ()**: vytvoří nový soubor a nastaví některé z atributů.
- ▶ **Delete ()**: smaže soubor.
- ▶ **Open ()**: do hlavní paměti se z FS načtou potřebné informace o souboru (atributy, diskové adresy,...).
- ▶ **Close ()**: uvolní se příslušné datové struktury v hlavní paměti, které související se souborem.
- ▶ **Read ()**: načte příslušný počet bytů od aktuální pozice.
- ▶ **Write ()**: zapíše příslušný počet bytů od aktuální pozice.
- ▶ **Seek ()**: nastaví aktuální pozici na novou hodnotu.
- ▶ **Funkce pro načtení atributů:** stat (), access (), ...
- ▶ **Funkce pro nastavení atributů:** chmod (), chown (), ...
- ▶ **Rename ()**: přejmenuje soubor.

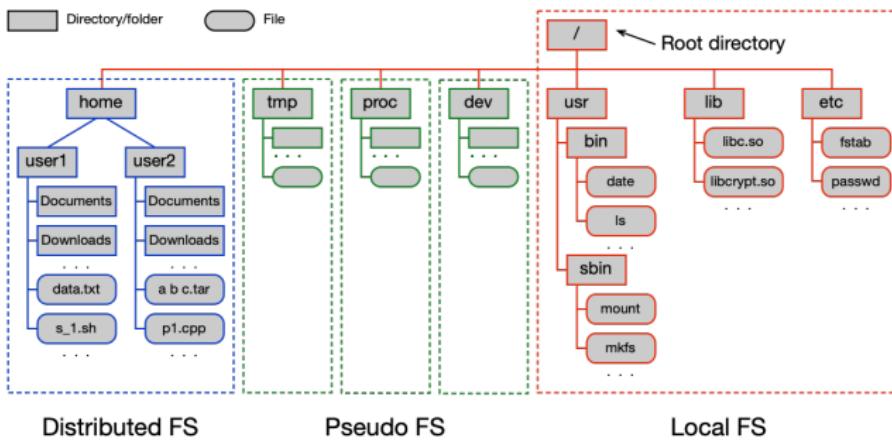
FS z pohledu administrátora

• Strom adresářů

- ▶ Celý strom adresářů může reprezentovat pouze jeden FS nebo několik FS propojených do hromady.

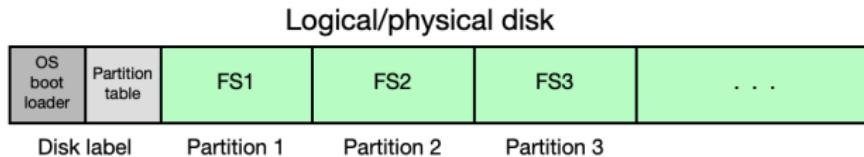
• Příklad stromu adresářů v OS unixového typu

- ▶ Typicky obsahuje několik FS
 - ★ Do kořenového adresáře je obvykle připojen lokální FS.
 - ★ Do podadresářů /proc, /tmp, /dev jsou připojeny příslušné pseudo FS (existují pouze v hlavní paměti, nikoliv na disku).
 - ★ Do podadresáře /home může být připojen distribuovaný FS (data jsou uložena na vzdáleném serveru).



FS z pohledu administrátora

• Rozložení dat na disku (disk layout)



- ▶ Logický/fyzický disk je typicky rozdělen na několik částí, které obsahují informace/data s různým významem pro OS.

1 Label disku (disk label)

- ★ Nachází se na začátku disku a obsahuje informace o rozdělení disku na jednotlivé oblasti (Partition table).
- ★ Může také obsahovat zavaděč OS (OS boot loader).
- ★ V praxi existuje několik různých formátů: MBR (Master Boot Record), EFI GPT (Extensible Firmware Interface GUID Partition Table),...

2 Jednotlivé diskové oblasti (partitions/slices)

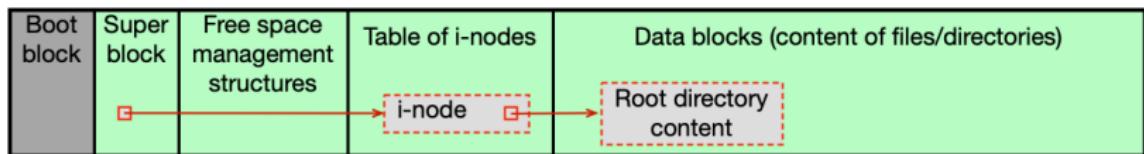
- ★ Každá disková oblast obsahuje jeden FS.
- Rozdelení disku na jednotlivé diskové oblasti, instalace zavaděče OS, vytvoření jednotlivých FS v příslušných oblastech se typicky provádí při instalaci systému a odpovídá za to administrátor systému. Pozdější změna rozložení dat na disku může být komplikovaná.

- Během instalace se také definuje, do kterých adresářů (přípojných bodů) ve stromu adresářů se budou jednotlivé FS připojovat. Toto lze relativně jednoduše upravit i později po instalaci systému.
- **Typické operace nad diskem a FS**
 - ▶ Rozdělení disku: např. příkazy fdisk (Linux), format (Solaris),...
 - ▶ Vytvoření FS: varianty unixových příkazů mkfs, mkfs.ext4, mkfs.vfat,...
 - ▶ Zvětšení FS: příkaz growfs (Solaris ufs),...
 - ▶ Kontrola FS: příkaz fsck,...
 - ▶ Připojení FS do stromu adresářů: příkaz mount
 - ▶ Odpojení FS: příkaz umount
 - ▶ Vytvoření zálohy FS: příkaz dump (Linux ext2/3/4), ufsdump (Solaris ufs),...
 - ▶ Obnova dat ze zálohy: příkaz restore (Linux ext2/3/4, Solaris ufs),...

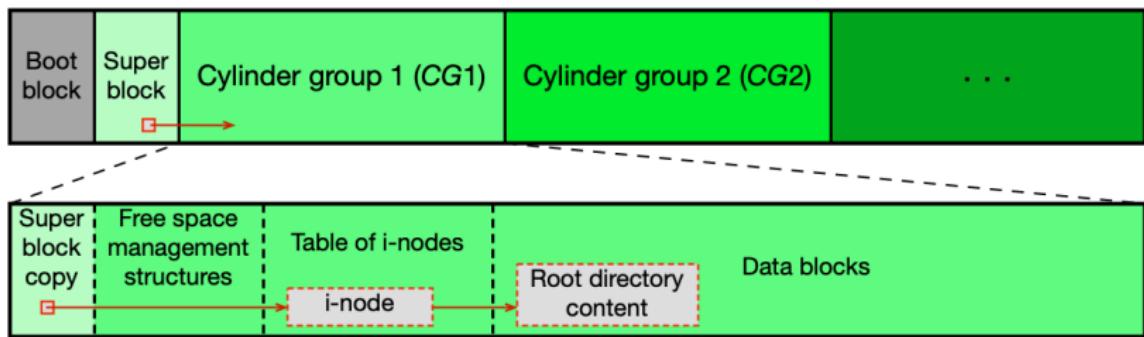
• Rozložení dat ve FS (FS layout)

- ▶ V oblasti disku, ve které je vytvořen FS, se obvykle nachází následující typy informací/dat.
 - ➊ Kód sloužící k **zavedení OS** (Boot block).
 - ➋ Informace popisující **konfiguraci FS** (Super block)
 - ★ Typ FS, Velikost datových bloků.
 - ★ Informace o celkové velikosti a aktuální obsazenosti FS (např. počet i-nodů, datových bloků,...).
 - ★ Informace o diskových adresách důležitých struktur FS.
 - ➌ Datové struktury pro **správu volného prostoru** (datových bloků, i-nodů,...).
 - ➍ Datové struktury pro uložení **atributů souborů** (Tabulka i-nodů,...).
 - ➎ **Datové bloky**, do kterých se ukládá obsah souborů a adresářů.

• Příklad rozložení dat v jednoduchém FS



• Příklad rozložení dat v UFS (Unix File System)



- ▶ UFS je reálný FS používaný např. v Solarisu nebo BSD.
- ▶ Z důvodu výkonu a zabezpečení dat proti ztrátě je diskový prostor UFS rozdělen do několika stejně velkých oblastí CG_i (Cylinder groups), které jsou reprezentovány souvislou množinou cylindrů na disku.
- ▶ Soubor/adresář je vždy alokován v rámci konkrétní $CG_i \Rightarrow$ lepší výkon (hlavičky HDD se pohybují pouze v rámci CG_i).
- ▶ Pokud dojde k poškození začátku disku (např. administrátor přepíše omylem Super blok a několik prvních sektorů z CG_1), pak se ztratí data z CG_1 , ale data z ostatních CG_i se podaří většinou zachránit pokud víme s jakými parametry byl UFS vytvořen.

- **Sector**

- ▶ Nejmenší adresovatelná jednotka datového úložiště (4 KB/512 B).

- **Datový blok**

- ▶ Velikost sektoru je obvykle příliš malá z hlediska FS a navíc je fixní pro dané datové úložiště.
 - ▶ Ve FS se proto používá alokace dat po větších **logických jednotkách**, které se nazývají **datové bloky** ("clusters" v MS Windows).
 - ▶ Velikost datového bloku může definovat administrátor při vytvoření FS a je závislá na velikosti samotného FS a na očekávané **velikosti ukládaných dat**.

- **Příklad**

- ▶ UFS (Unix file system) v Solarisu má pouze 8KB datové bloky.
 - ▶ Unixový VXFS (Veritas FS) má 1, 2, 4, 8 KB datové bloky.
 - ▶ FAT32 v MS Windows má 4KB,...,32KB datové bloky.
 - ▶ NTFS v MS Windows má 4KB až 2 MB datové bloky.

Implementace FS

• Správa volných datových struktur

- ▶ Podobně jako při správě paměti i zde lze použít pro správu volných datových struktur buď bitovou mapu nebo zřetězený seznam.
- ▶ **Zřetězený seznam**
 - ★ Je uložen přímo ve volných datových blocích FS.
 - ★ Po připojení FS je obvykle nahrána do hlavní paměti pouze část tohoto seznamu.
- ▶ **Bitová mapa**
 - ★ Většinou zabírá méně místa než zřetězený seznam.
 - ★ Pouze v případě téměř zaplněného FS je zřetězený seznam výhodnější z hlediska velikosti.



● Obsah souborů/adresářů je uložen v datových blocích

- ▶ Z hlediska výkonu FS je klíčový způsob alokace datových bloků.
- ▶ Rozlišujeme dva základní přístupy
 - 1 alokace souvislé oblasti datových bloků (contiguous run of blocks),
 - 2 alokace po jednotlivých datových blocích.

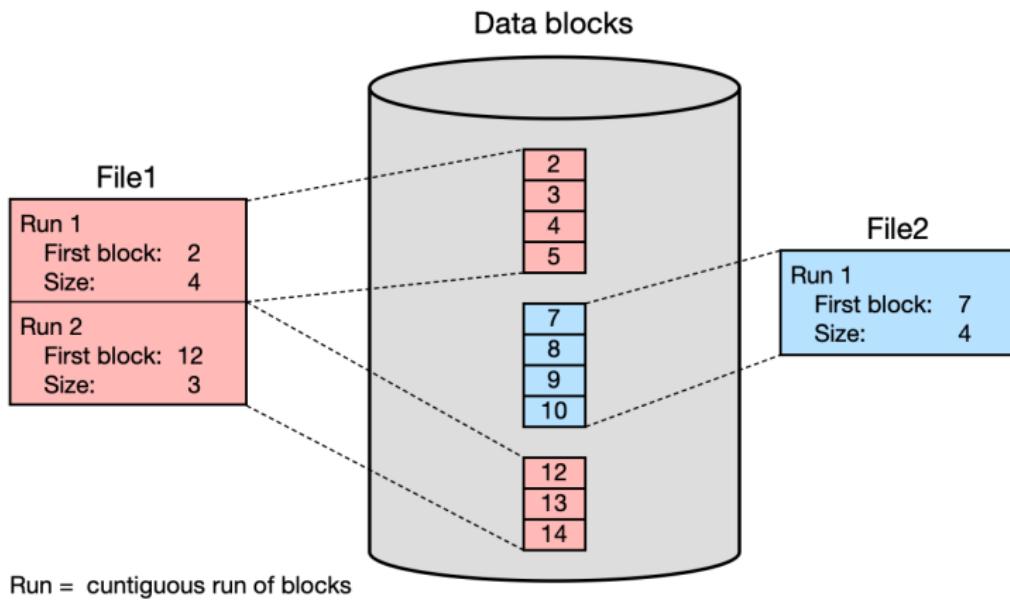
1 Alokace souboru pomocí souvislých oblastí datových bloků

- ▶ V ideálním případě je obsah souboru uložen v jedné souvislé oblasti bloků. Protože se v čase velikost souboru většinou mění, tak častěji je jeho obsah uložen v několika souvislých oblastech bloků.
- ▶ **Výhody**
 - ★ Pro každý soubor si FS musí pamatovat malý počet informací (např. adresu prvního bloku a počet bloků).
 - ★ Výborný výkon při sekvenčním přístupu a u velkých souborů.
- ▶ **Nevýhody**
 - ★ Problém s fragmentací FS
 - ⇒ složitější alokace souvislé oblasti,
 - ⇒ je nutné pravidelně defragmentovat.
- ▶ **Příklady**
 - ★ VXFS (Veritas File System),
 - ★ NTFS (New Technology File System),...

Implementace FS

• Příklad: Alokace obsahu souboru pomocí souvislý oblastí

- ▶ Obsah souboru File1 je uložený ve dvou souvislých oblastech, které se skládají z bloků 2, ..., 5 a 12, ..., 13.
- ▶ Obsah souboru File2 je uložený v jedné souvislé oblasti, která je složena z bloků 7, ..., 10



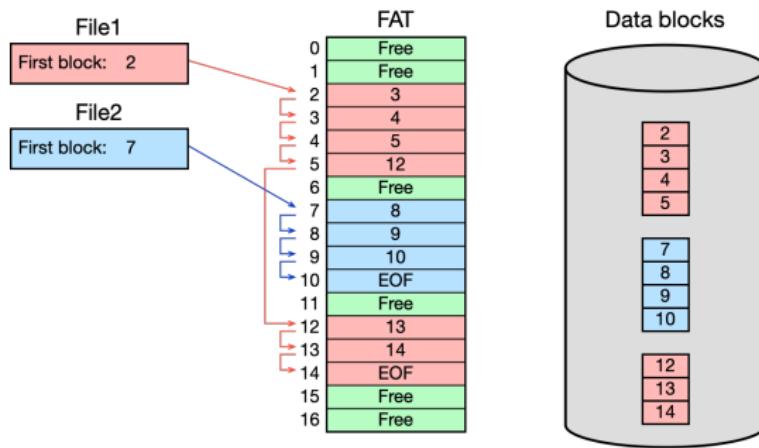
2 Alokace souboru po jednotlivých datových blocích

- ▶ Obsah souboru je alokován po jednotlivých blocích
⇒ FS si musí pamatovat adresy všech bloků, kde je uložený obsah souboru.
- ▶ Pro uložení adres bloků se používají následující struktury
 - a FAT (File Allocation Table),
 - b i-node (index node).
- ▶ Přestože se obsah souboru alokuje po jednotlivých blocích, ovladač FS se snaží uložit obsah souboru v rámci jednoho nebo několika sousedních cylindrů tak, aby se minimalizoval čas vystavení hlaviček u HDD.
- ▶ **Výhody**
 - ★ Není problém s fragmentací.
 - ★ Dobrý výkon při náhodném přístupu a u malých souborů.
- ▶ **Nevýhody**
 - ★ Pro každý soubor si FS musí pamatovat velký počet informací.
 - ★ Horší výkon při sekvenčním přístupu.
- ▶ **Příklady**
 - ★ FAT32 (File Allocation Table),
 - ★ UFS (Unix File System),
 - ★ EXT2/3/4 (Extended File System),...

Implementace FS

• FAT (File Allocation Table)

- ▶ umožňuje alokaci obsahu souboru po jednotlivých datových blocích.
- ▶ Tabulka obsahuje kolik je datových bloků ve FS.
- ▶ Pro každý soubor si musíme pamatovat pouze adresu prvního bloku, adresy dalších bloků jsou uloženy ve FAT formou zřetězení.
- ▶ Každý řádek i tabulky obsahuje právě jednu z následujících hodnot
 - ★ Free: datový blok i je volný,
 - ★ Adresa: adresa následujícího bloku za blokem i , kde pokračuje obsah souboru,
 - ★ EOF: konec zřetězení (blok i je posledním blokem souboru).



Implementace FS

● Vlastnosti FAT

- ▶ Informace o volných datových blocích se dají vyčíst přímo z FAT.
- ▶ Při připojení systému s FAT se do hlavní paměti načte celá/část FAT \Rightarrow urychlí se přístup k obsahu souboru.
- ▶ Pro velké disky vzniká **problém s velikostí FAT**, protože velikost datových bloků se v čase příliš nenavyšuje.

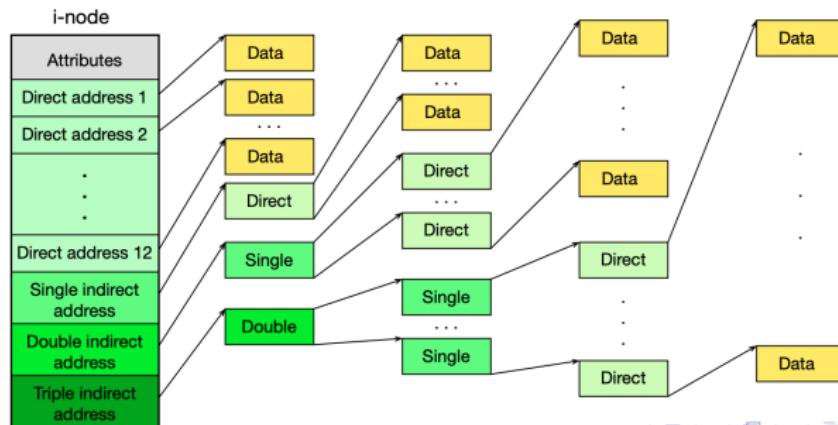
● Příklad

- ▶ Předpokládejme následující parametry
 - ★ Velikost datových bloků je 1KB.
 - ★ Velikost adresy je 32 bitů.
- ▶ Z těchto parametrů vyplývají následující vlastnosti.
 - ★ FAT může mít až 2^{32} řádek.
 - ★ Jedna řádka zabírá 32 bitů = 4 B.
 - ★ Velikost FAT může být až $2^{32} \times 4B = 2^{34} = 16GB$.
 - ★ Maximální velikost FS může být teoreticky až $2^{32} \times 1KB = 2^{42}B = 4TB$.
 - ★ Maximální velikost souboru může být teoreticky až $2^{32} \times 1KB = 2^{42} = 4TB$.
 - ★ **POZOR:** maximální velikosti a maximální počty mohou být omezeny velikostí ostatních struktur ve FS nebo OS.

Implementace FS

● I-node (Index node)

- ▶ S každým souborem/adresářem je spojen příslušný i-node.
- ▶ I-node je datová struktura, která má **fixní velikost** a ve které jsou **uloženy atributy souboru/adresáře** a **adresy datových bloků**, kde je uložen jeho obsah.
- ▶ **Z důvodu adresace různě velkých souborů jsou zde uloženy tři typy adres**
 - ★ **12 přímých adres:** ukazující přímo na datové bloky, kde je obsah souboru,
 - ★ **1 nepřímá adresa první úrovně:** ukazuje na blok, ve kterém jsou přímé adresy,
 - ★ **1 nepřímá adresa druhé úrovně:** ukazuje na blok, ve kterém jsou nepřímé adresy první úrovně,
 - ★ **1 nepřímá adresa třetí úrovně:** ukazuje na blok, ve kterém jsou nepřímé adresy druhé úrovně.



Implementace FS

● Vlastnosti i-nodů

- ▶ Při otevření souboru/adresáře se do hlavní paměti načte pouze jeho i-node a teprve při čtení/zápisu se načítají jednotlivé bloky s daty/adresami.
- ▶ Při zápisu do souboru se **postupně využívají přímé adresy, nepřímé adresy první, druhé a nakonec třetí úrovně** v závislosti na velikosti souboru.
- ▶ U velkých souborů a náhodném přístupu je **pomalejší přístup k datům** při první přístupu. Při následujících přístupech se již využívá skrytá paměť.
- ▶ Horší využití prostoru **FS**, protože část datových bloků se používá na metadata (bloky s adresami).
- ▶ Samotná velikost i-nodu není závislá na velikosti FS nebo velikosti souboru.
- ▶ Počet i-nodů se implicitně odvozuje od kapacity FS. V řadě FS je počet i-nodů statický (po vytvoření FS nelze počet navýšit), např. UFS, EXT2/3/4, VxFS,...

● Příklad

- ▶ Předpokládejme následující parametry
 - ★ Velikost datových bloků je 4KB.
 - ★ Velikost adresy je 32 bitů.
- ▶ Z těchto parametrů vyplývají následující vlastnosti.
 - ★ Počet adres v bloku je $4KB/32bit = 2^{10}$.
 - ★ Maximální velikost souboru může být teoreticky
$$(12 + 2^{10} + 2^{10} \times 2^{10} + 2^{10} \times 2^{10} \times 2^{10}) \times 4KB \approx 2^{30} \times 4KB = 4TB$$
 - ★ **POZOR:** maximální velikosti a maximální počty mohou být omezeny velikostí ostatních struktur ve FS nebo OS.

Implementace FS

• Adresáře

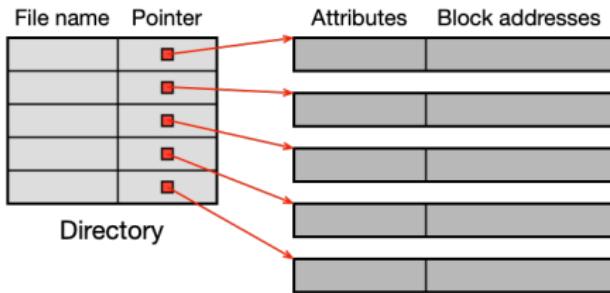
- ▶ Obsah adresářů je podobně jako obsah souborů uložen v jednom nebo několika datových blocích.
- ▶ Z hlediska implementace existují dva přístupy.
 - ★ Adresář obsahuje **většinu informací** o souborech a podadresářích (jméno, atributy a adresy bloků s obsahem souboru), např. FAT32.
 - ★ Adresář obsahuje **minimum informací** (jméno a odkaz do "speciální" datové struktury (např. i-node v UFS nebo položka Master File Table v NTFS), ve které jsou uloženy ostatní informace).

Directory with maximum information

File name	Attributes	Block addresses

Directory

Directory with minimum information

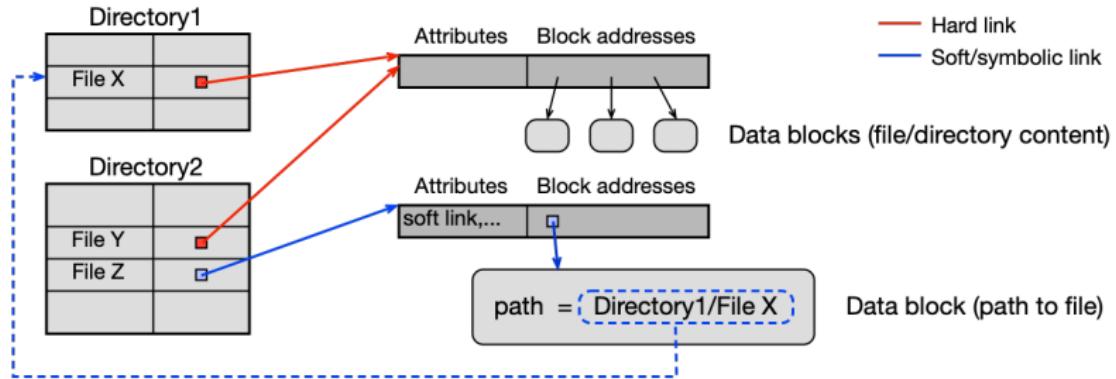


Directory

i-node/MFT entry

• Sdílené soubory

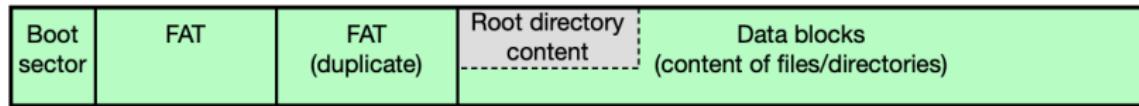
- ▶ Občas je vhodné, aby **ten samý soubor/adresář** byl současně viditelný v různých adresářích (popř. i pod různými jmény).
- ▶ Většina současných FS toto umožňuje a k dispozici jsou dvě implementace.
 - ★ **Soft link:** odkaz na existující soubor/adresář je implementován pomocí cesty k souboru.
 - ★ **Hard link:** implementován pouze u minimalistických adresářů přímo pomocí odkazu na "speciální datovou strukturu" (i-node/MFT položku).



Příklad: FAT32/exFAT

• Popis

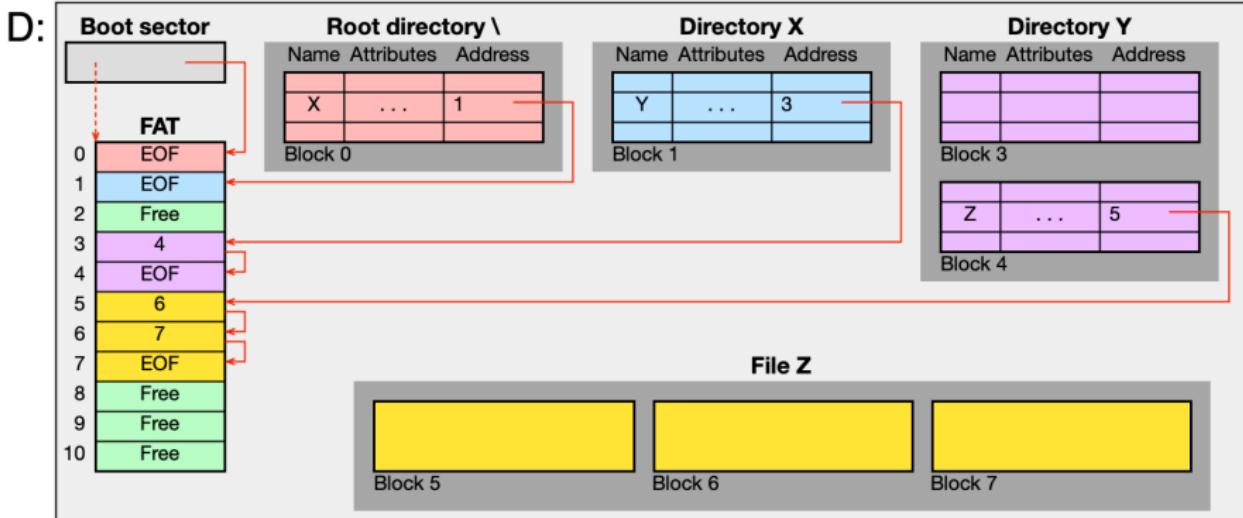
- ▶ Existuje několik různých implementací systém souborů FAT podle velikosti adresy
 - ★ FAT12 (12 bitová verze),
 - ★ FAT16 (16 bitová verze),
 - ★ **FAT32** (32 bitová verze používající 28 bitů),
 - ★ **exFAT** (64 bitová verze).
- ▶ Obsah adresáře je implementován jako tabulka a je uložen v jednom nebo několika datových blocích.
- ▶ Záznam o souboru/podadresáři je obvykle uložen v jedné řádce tabulky, která má statickou strukturu a obsahuje
 - ★ jméno souboru,
 - ★ atributy souboru (typ, velikost,...),
 - ★ adresu prvního bloku, kde začíná obsah souboru/podadresáře (informace o dalších blocích, kde pokračuje obsah, jsou ve FAT).
- ▶ Pokud se některé informace (jméno souboru, ACL práva,...) nevejdou do vyhrazeného místa mohou být uloženy ve více řádkách adresáře.
- ▶ **Rozložení dat na disku u FAT32**
 - ★ exFAT obsahuje navíc bitovou mapu s informací o volných datových blocích.



Příklad: FAT32/exFAT

• Přístup k adresáři/souboru

- ▶ Boot sektor obsahuje informaci, kde začíná FAT, kopie FAT a obsah kořenového adresáře (Root directory).
- ▶ Při připojení FS se načte do hlavní paměti celé/část FAT.
- ▶ Pokud chceme např. zobrazit obsah souboru D:\X\Y\Z, pak musíme načíst z disku příslušné datové bloky.

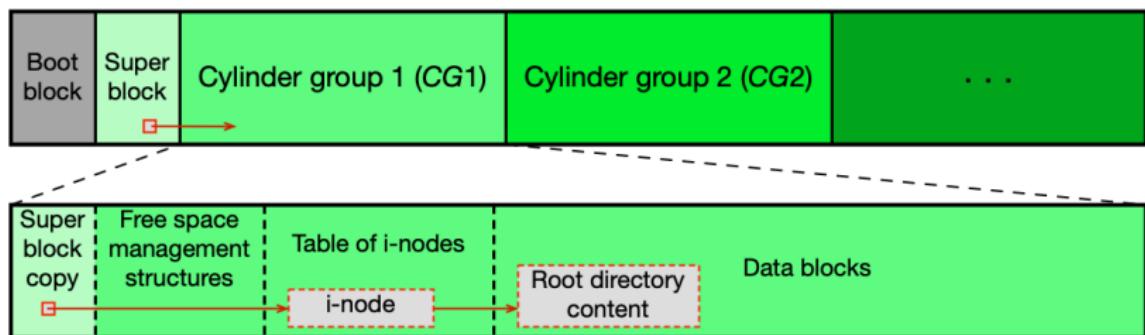


Příklad: UFS (Unix File System)

• Popis

- ▶ Někdy je také označovaný jako BSD Fast File System (FFS).
- ▶ Můžeme ho najít v různých OS (BSD, Solaris,...) a v různých verzích, ve kterých jsou implementovány nové vlastnosti (např. žurnálování, snapshoty, ACL práva,...).
- ▶ Byly jím inspirovány další FS (HFS+ v MACOS, Ext2/3/4 v Linuxu,...).
- ▶ V OS Solaris je velikost datového bloku 8KB, adresář obsahuje jméno souboru a číslo i-node daného souboru/podadresáře.
- ▶ i-node má velikost 128 B a obsahuje atributy souboru/adresáře (přístupová práva, vlastníka, ...) a 15 32-bitových adres diskových bloků (12 přímých a 3 nepřímé adresy první, druhé a třetí úrovně).

• Rozložení dat na disku

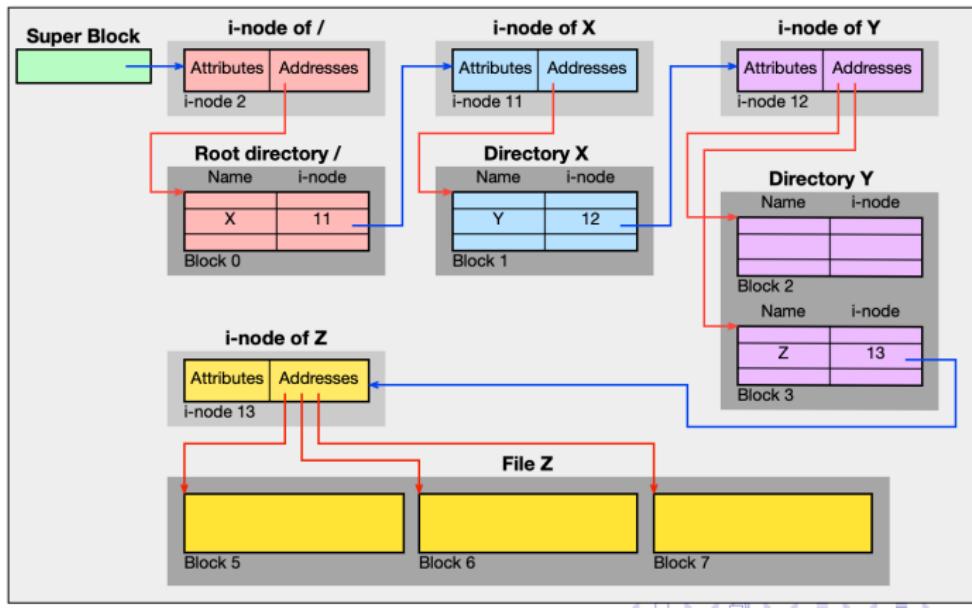


Příklad: UFS (Unix File System)

• Přístup k souborům/adresářům

- ▶ Super blok obsahuje informaci, kde začínají struktury pro správu volného prostoru, tabulka i-nodů a datové bloky.
- ▶ Po připojení UFS se do paměti načte i-node kořenového adresáře (i-node číslo 2).
- ▶ Pokud chceme např. zobrazit obsah souboru /X/Y/Z, pak musíme načíst z disku příslušné i-nody a datové bloky.

Root FS:



- ① A. S. Tanenbaum, H. Bos: *Modern Operating Systems (4th edition)*, Pearson, 2014.
- ② W. Stallings: *Operating Systems: Internals and Design Principles (9th edition)*, Pearson, 2017.
- ③ A. Silberschatz, P. B. Galvin, G. Gagne: *Operating System Concepts (9th edition)*, Wiley, 2012.
- ④ R. McDougall, J. Mauro: *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture (2nd edition)*, Prentice Hall, 2006.

Operační systémy

Systémy souborů II

Jan Trdlička



České vysoké učení technické v Praze, Fakulta informačních technologií
Katedra počítačových systémů

<https://courses.fit.cvut.cz/BI-OSY>

Obsah přednášky

1

Implementace FS v OS

- Virtuální FS/Installable FS
- Filesystem in userspace (FUSE)
- Block cache/Page cache
- Directory Name Look-up Cache(DNLC)
- Block Read Ahead
- Žurnálované FS

2

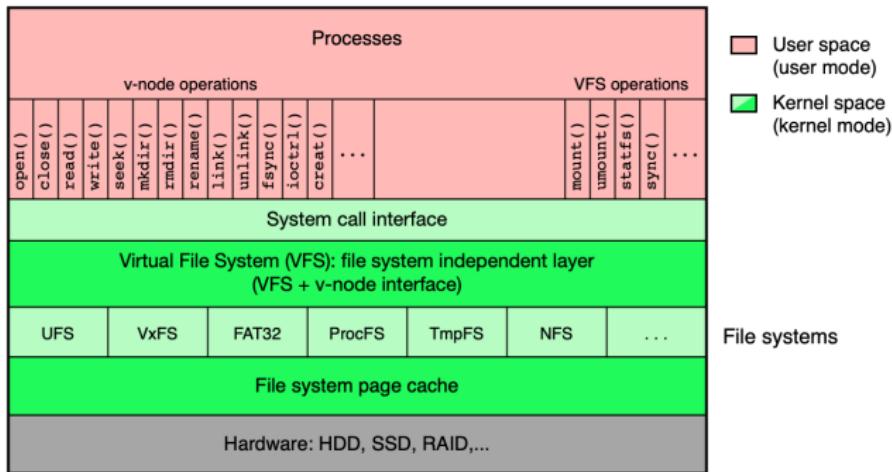
Moderní FS

- B stromy a B+ stromy
- Vlastnosti FS
 - ZFS (Zettabyte FS)
 - BTRFS (B-tree FS)

Implementace FS v OS

● Virtual File System (VFS)

- ▶ Framework v rámci, kterého jsou v OS unixového typu implementovány konkrétní systémy souborů.
- ▶ Vrstva jádra OS, která představuje rozhraní mezi procesy a jednotlivými implementacemi konkrétních FS.



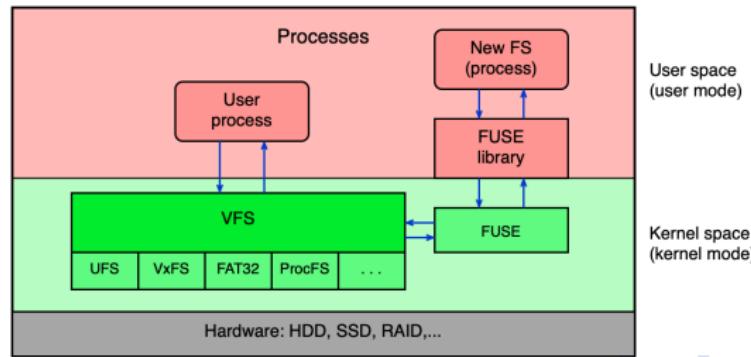
● Installable File System (IFS)

- ▶ API v IBM OS/2 a v MS Windows, které umožňuje OS rozpoznat a nahrát ovladač pro konkrétní FS.

Implementace FS v OS

● Filesystem in Userspace (FUSE)

- ▶ Rozhraní v OS unixového typu, které umožňuje neprivilegovaným uživatelům vytvářet své vlastní FS bez nutnosti modifikovat jádro OS.
- ▶ FUSE obsahuje
 - ★ modul jádra OS: samotná implementace FUSE,
 - ★ knihovna: rozhraní k modulu.
- ▶ Nový FS implementovaný pomocí FUSE
 - ★ Implementován jako proces, ve kterém se musíme definovat standardní operace nad novým FS.
 - ★ Běžný uživatel může manipulovat s tímto FS pomocí standardních knihovních funkcí/příkazů.
- ▶ Vhodný pro implementaci pseudo FS (FS, který existuje pouze v hlavní paměti) a data uložit prostřednictvím existujících FS.



Implementace FS v OS

- Přístup na disk je mnohem pomalejší než přístup do hlavní paměti
 - ⇒ OS optimalizuje přístup do FS různými způsoby
 - ▶ Využívá volnou hlavní paměť jako **skrytou paměť (cache)** pro
 - ★ **datové bloky FS** (např. Block cache/Page cache),
 - ★ **metadata FS** (např. Directory Name Look-up Cache).
 - ▶ Používá **přednačítání datových bloků FS** (např. Block Read Ahead) na základě principu prostorové lokality při přístupu k datům.
 - **Block cache/Page cache**
 - ▶ Velikost datových bloků FS je obvykle navržena tak, aby byla **násobkem velikosti stránek ve virtuální stránkovane paměti**.
 - ▶ Block cache/Page cache je **množina nedávno používaných datových bloků z FS (obsah souborů/adresářů)** uložená v hlavní paměti z důvodu zlepšení výkonu FS.
 - ▶ OS si udržuje informaci o všech blocích načtených do hlavní paměti.
 - ▶ Když proces **čte obsah** adresáře/souboru, OS hledá bloky (obsah) nejdříve v hlavní paměti, pokud zde nejsou, pak je načte z FS.
 - ▶ Když proces **modifikuje obsah** adresáře/souboru, pak existují dva přístupy
 - ★ **write-behind cache**: modifikace je zapsána ihned do datového bloku v hlavní paměti a se zpožděním do FS (běžně používané u FS na interních datových úložištích),
 - ★ **write-through cache**: modifikace je současně zapsána do hlavní paměti i do FS (běžně používané u FS na přenosných datových úložištích typu flash-disk).

Implementace FS v OS

● Příklad: Page cache v Solarisu

- ▶ Informaci o aktuálním obsazení fyzické můžeme získat v OS Solaris pomocí modulárního debagraru jádra prostřednictvím příkazu mdb -k.

Page Summary	Pages	Bytes	%Tot
Kernel	115198	449.9M	22%
ZFS Metadata	18159	70.9M	3%
ZFS File Data	135738	530.2M	26%
Anon	102981	402.2M	20%
Exec and libs	3869	15.1M	1%
Page cache	26831	104.8M	5%
Free (cachelist)	3	12k	0%
Free (freelist)	101709	397.3M	19%
Total	524159	1.9G	

- ▶ V systému jsou používány dva systémy souborů: ZFS a UFS.
- ▶ Ve výpisu vidíme následující informace

- ★ Kernel: paměti alokovaná jádru OS.
- ★ ZFS Metadata: skrytá paměť systému souborů ZFS pro metadata
- ★ ZFS File Data: skrytá paměť systému souborů ZFS pro datové bloky.
- ★ Anon: anonymní paměť (stránky procesů, které nesouvisí s FS, např. zásobníky, haldy,...).
- ★ Exec and libs: bloků binárních programů a knihoven.
- ★ Page cache: skrytá paměť systému souborů UFS pro datové bloky.
- ★ Free (cachelist): volné stránky s použitelným obsahem (např. soubory, které aktuálně nejsou nikým používány, ale v budoucnu by opět mohly být použity,...).
- ★ Free (freelist): volné stránky bez použitelného obsahu (např. zásobníky, haldy ukončených procesů,...).

Implementace FS v OS

● DNLC (Directory Name Look-up Cache)

- ▶ Skrytá paměť v OS Solaris, která obsahuje jména nedávno používaných souborů/adresářů a jejich v-nody (datova struktura VFS).
- ▶ Při následujícím přístupu k souboru/adresáři se použije informace z DNLC, nikoliv z datových bloků FS.

● Příklad: DNLC v Solarisu

- ▶ Pokud vytvoříme adresáře A/B/C/D/E/F/G/H/I/J/K/L a restartujeme OS, aby se vyčistila DNLC

```
user@solaris:~$ mkdir -p A/B/C/D/E/F/G/H/I/J/K/L
```

```
user@solaris:~$ sudo reboot
```

- ▶ Pak můžeme zjistit vliv DNLC na rychlosť přístupu k souborů/adresářům.

```
user@solaris:~$ time ls -l A/B/C/D/E/F/G/H/I/J/K/L
total 0
```

```
real      0m0.015s
user      0m0.001s
sys       0m0.012s
```

```
user@solaris:~$ time ls -l A/B/C/D/E/F/G/H/I/J/K/L
total 0
```

```
real      0m0.002s
user      0m0.000s
sys       0m0.001s
```

Implementace FS v OS

● Přednačítání datových bloků FS (Block Read Ahead)

- ▶ Předpokládá se, že pro přístup k datům platí princip prostorové lokality.
- ▶ Při požadavku čtení konkrétního datového bloku se z FS do paměti načte současně tento blok a několik následujících bloků.
- ▶ Tímto přístupem lze zlepšit např. sekvenční čtení nebo využít datového úložiště typu RAID.

● Příklad: Block Read Ahead v UFS

- ▶ UFS má velikost bloku FS fixně nastavenou na 8 kB a odpovídá velikosti stránky na CPU SPARC.
- ▶ Parametr UFS **maxcontig** definuje, kolik bloků se bude číst/zapisovat při jedné operaci čtení/zápisu.
- ▶ Příkaz **fstyp** v OS Solaris vypíše aktuální hodnoty parametrů UFS.

```
root@solaris:~> fstyp -v /dev/rdsck/c2t1d0s2 | head
ufs
magic 11954    format dynamic time   Thu May  9 16:21:46 2019
sblkno 16       cbblkno 24      ibblkno 32      dblkno 760
sbsize 2048    cgsizze 8192    cgoffset 64      cgmask 0xffffffffc0
ncg 213        size 10461696   blocks 10303204
bsize 8192     shift 13      mask 0xfffffe000
fsize 1024     shift 10      mask 0xfffffff0c0
frag 8         shift 3       fsbtodb 1
minfree 1%     maxbpg 2048    optim time
maxcontig 32  rotdelay 0ms    rps 120
...
```

- ▶ Novou hodnotu parametru **maxcontig** může administrátor nastavit příkazem **tunefs**.

```
root@solaris:~> tunefs -a 128 /dev/rdsck/c2t1d0s2 | head
```



● Ochrana dat při pádu systém/výpadku napájení

- ▶ Vytvoření/modifikace souboru/adresáře se obvykle skládá z několika kroků.
 - ★ Modifikace struktur související s volným prostorem (i-nody, datové bloky,...).
 - ★ Modifikace obsahu adresáře, ve kterém se soubor/adresář má vytvořit.
 - ★ Modifikace metadat (položky adresáře, i-nodu/položky v MFT,...).
 - ★ Modifikace dat v datových blocích.
- ▶ Pokud všechny tuto kroky nejsou dokončeny, FS může zůstat v nekonzistentním stavu.
- ▶ Při následné opravě FS (např. pomocí příkazu `fsck` v OS unixového typu) můžeme o některá data přijít.

● Žunrálovaný FS (Journaling FS)

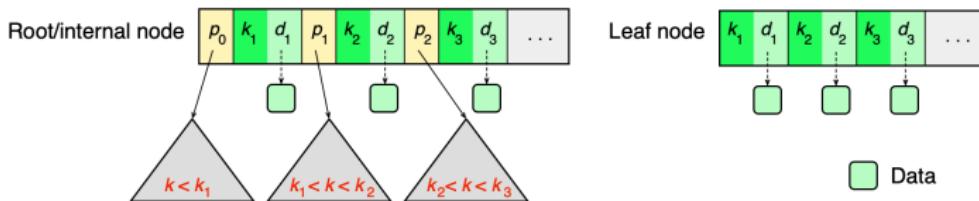
- ▶ Snaží se ochránit FS před nekonzistencí a ztrátou dat.
- ▶ FS si alokuje na disku speciální oblast "journal", do které si v předstihu zapíše záznam o změnách, které se budou následně provádět. Potom, co se změny úspěšně zapíší do FS, je záznam z "journalu" odstraněn.
- ▶ V případě havarie, po zotavení systému se "přehrají" záznamy z "journalu" a FS vrátí do konzistentního stavu.
- ▶ "Journaling" je implementován ve většině současných FS (např. UFS, EXT2/3/4, NTFS,...).
- ▶ Z důvodu výkonu se do "journálu" většinou ukládají pouze metadata ale nikoliv obsah datových bloků.
- ▶ "Journal" s nemusí vždy nacházet na stejném disku jako samotný FS, ale může být umístěn např. na rychlejším SSD.

- Většina klasických FS (např. UFS, EXT2/3/4, FAT32, NTFS,...) byla navržena v 80. a 90. letech minulého století.
- Díky vývoji datových úložišť (zvláště RAID, SSH), navýšení diskových kapacit a CPU výkonu systémů začaly vznikat moderní FS jako např. **ZFS (Zettabyte FS)**, **BTRFS (B-tree FS)**, **APFS (Apple FS)**,...
- Mezi jejich důležité vlastnosti patří
 - ▶ Při implementaci FS se používají B stromy/B+ stromy.
 - ▶ Většinou reprezentují kombinaci SW RAIDu a FS.
 - ▶ Integrita dat (FS je neustále v konzistentním stavu).
 - ★ Redundance pomocí RAID \Rightarrow ochrana proti výpadkům HW komponent.
 - ★ Copy-on-write transactional object model \Rightarrow ochrana dat proti výpadku napájení,....
 - ★ Kontrolní součty dat/metadat \Rightarrow detekce a oprava dat/metadat.
 - ▶ Podpora efektivního vytváření "clonů" a "snapshotů".
 - ▶ Podpora šifrování dat.
 - ▶ Podpora kompresi dat a deduplikaci datových bloků.
 - ▶ Jednoduší administrace FS (obvykle se vystačí pouze s několika příkazy, např. `zpool` a `zfs` v ZFS).

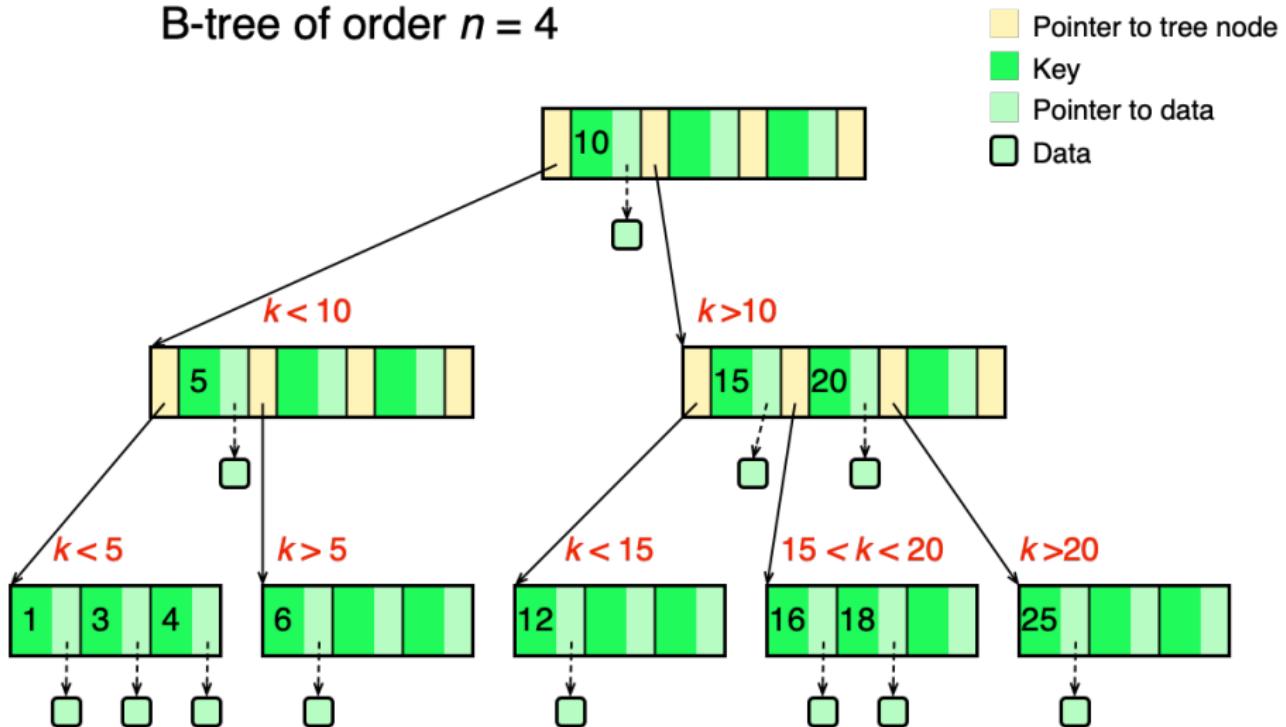
Moderní FS

• B strom řádu n

- ▶ Kořen stromu
 - ★ Má nejméně 2 a nejvíce n potomků, pokud není listem.
 - ★ Obsahuje nejméně 0 a nejvýše $n - 1$ položek.
- ▶ Vnitřní uzel stromu
 - ★ Má nejméně $\lceil n/2 \rceil$ a nejvýše n potomků.
 - ★ Obsahuje nejméně $\lceil n/2 \rceil - 1$ a nejvýše $n - 1$ položek.
- ▶ List stromu
 - ★ Všechny cesty od kořene k listům jsou stejně dlouhé.
 - ★ Obsahuje nejméně $\lceil n/2 \rceil - 1$ a nejvýše $n - 1$ položek.
- ▶ Data v uzlech stromu tvoří uspořádanou posloupnost
 - ★ $p_0, [k_1, d_1], p_1, [k_2, d_2], p_2, \dots$ (kořen/vnitřní uzel),
 - ★ $[k_1, d_1], [k_2, d_2], \dots$ (list stromu),
- ▶ Pro každé k_i označují klíčové atributy ukládaných dat d_i , takové, že pro ně jde zavést relace uspořádání $<$ a p_i označují odkazy na podstromy. Pro každé k_i a k_j , kde $i < j$ je $k_i < k_j$.
- ▶ Pro každé k v podstromě odpovídajícím odkazu p_{i-1} platí $k < k_i$.
- ▶ Pro každé k v podstromě odpovídajícím odkazu p_i platí $k > k_i$.

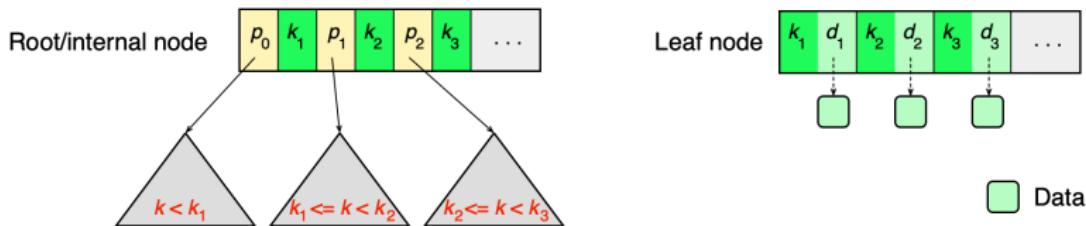


B-tree of order $n = 4$

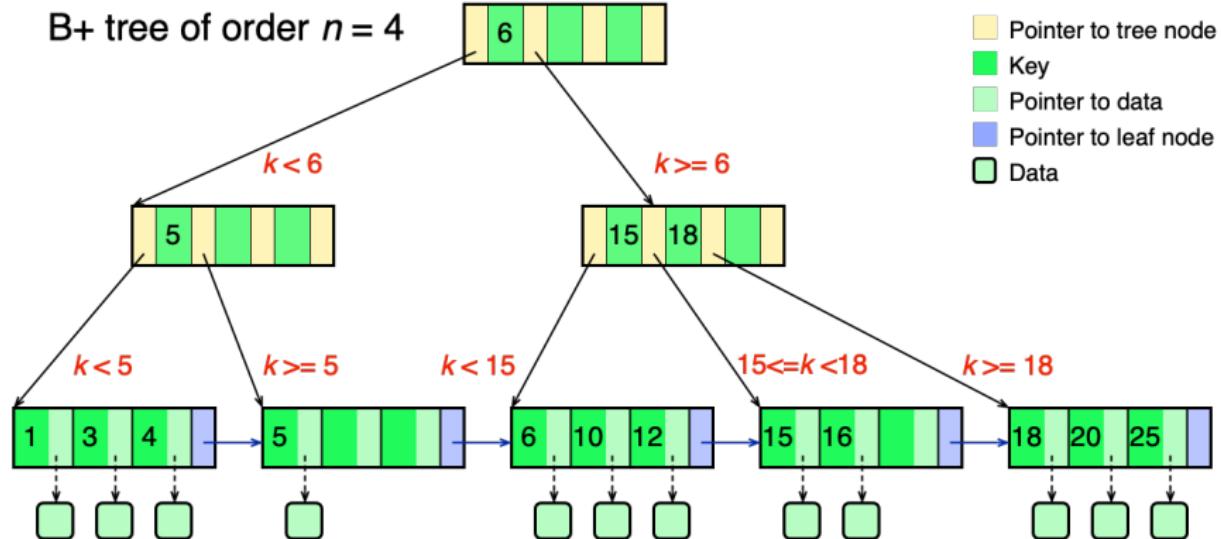


● B+ stromy

- ▶ Hlavní rozdíl od B stromů je, že **v kořenu/vnitřních uzlech jsou uloženy pouze klíče a odkazy na podstromy a v listech stromu jsou uloženy klíče s daty**. Ostatní vlastnosti má stejné jako zmínovaný B-strom.
- ▶ B+ strom se ve skutečnosti realizuje tak, že je vždy ve všech listech uložen kromě vlastních klíčů a dat také odkaz (ukazatel) na následující sourozence.



Moderní FS

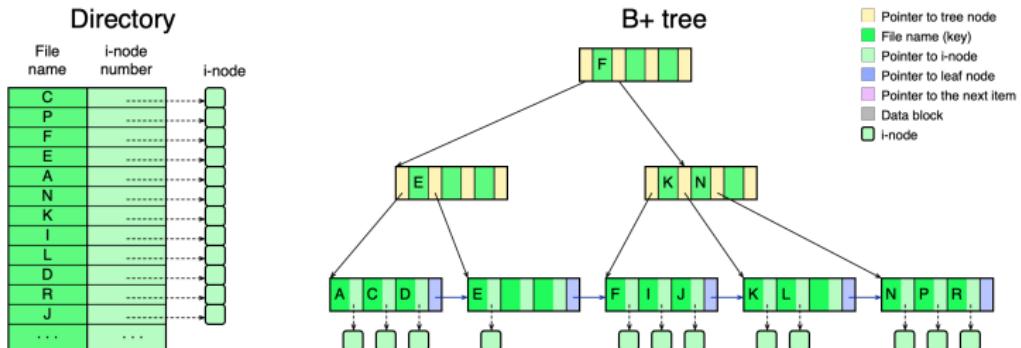


● Vlastnosti B stromů/B+ stromů

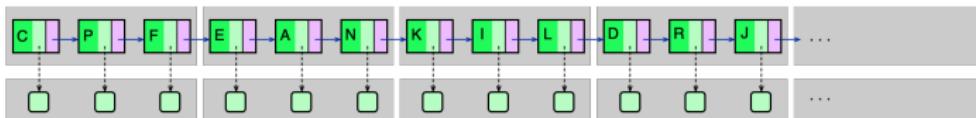
- ▶ Vložení/smazání/hledání prvku má časovou složitost $O(\log(N))$, kde N je počet záznamů ve stromě.
- ▶ V uzlech je uloženo více klíčů
⇒ uzel může být uložen ve skryté paměti L1/L2/L3 jako blok, v hlavní paměti jako stránka, na disku/FS jako sektor/datový blok.
- ▶ Uzly s klíči mohou být nahrány do paměti, zatímco data zůstávají na disku.

Moderní FS

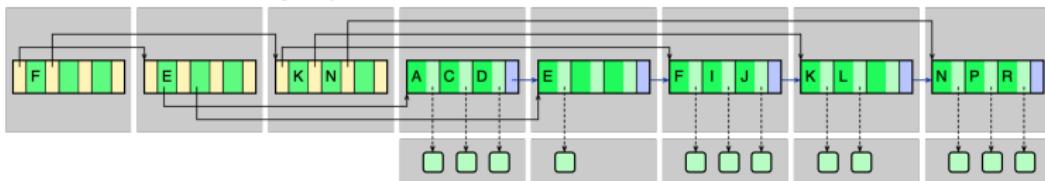
• Příklad: Použití B+ stromů při implementaci adresářů



Directory implemented like link list inside data blocks



Directory implemented like B+ tree inside data blocks



Moderní FS

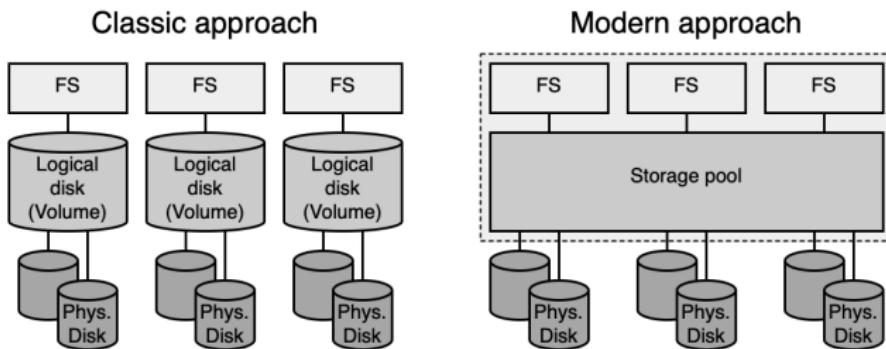
• Kombinace SW RAIDu a FS

► Klasický přístup

- ★ SW RAID a FS jsou oddělené (dvě nezávislé SW vrstvy).
- ★ V SW RAIDu se vytvoří z fyzických disků logický disk (Volume) s příslušnými vlastnostmi a v něm se následně vytvoří příslušný FS.

► Moderní přístup

- ★ SW RAID a FS jsou implementovány jako celek (jedna SW vrstva).
- ★ Fyzické disky se zařadí do "poolu", který představuje konkrétní typ RAIDu.
- ★ V rámci poolu se pak vytváří jednotlivé FS.
 - ⇒ Efektivnější využívání a sdílení kapacity fyzických disků.
 - ⇒ Jednodušší administrace (zvětšování/zmenšování jednotlivých FS).

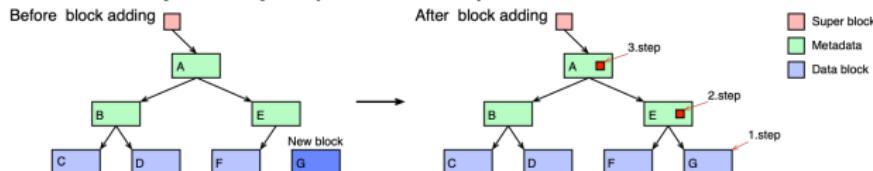


Moderní FS

• Integrita dat

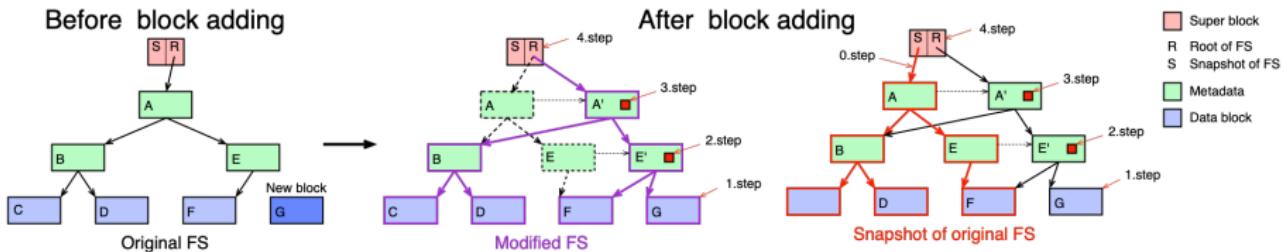
► Klasický přístup

- ★ Při změně dat/metadata ve FS se původní data přepíší novou hodnotou.
- ★ Změna probíhá v několika krocích (např. zvětšení obsahu souboru o blok G
⇒ zápis bloku G, zápis metadata A a E).
- ★ Pokud všechny změny neproběhnou, pak FS skončí v nekonzistentním stavu.



► Copy-On-Wire (COW) přístup

- ★ Původní data se nepřepisují, ale vytvoří se kopie, která se teprve modifikuje.
- ★ Po provedení všech změn se atomicky přepíše ukazatel v Super bloku
⇒ FS je neustále konzistentní.



Moderní FS

• Různé typy poškození dat

- 1 Bit corruption: Původní obsah sektoru/datového bloku byl modifikován (např. elektromagnetickým zářením,...). Data byla zapsána do sektoru/bloku, který je poškozený (obsah bloku se liší od zapisovaných data).
- 2 Lost writes: Disková operace "write" se nepovedla, ale úspěšné dokončení bylo potvrzeno.
- 3 Misdirected writes: Data byla zapsána do špatného datového bloku.
- 4 Torn writes: Data byla zapsána pouze částečně, ale úplné dokončení bylo potvrzeno.

• Zabezpečení dat

► Klasický přístup

- ★ Klasické FS se spoléhají pouze na kontrolní součty (ECC), které zabezpečují obsah sektoru. V lepším případě používají ještě kontrolní součty v rámci FS na zabezpečení obsahu datových bloků.
- ★ Kontrolní součty jsou typicky umístěny společně s daty v sektoru/bloku.
⇒ Detekuje/opráví pouze poškození dat typu 1.

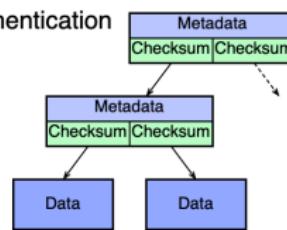
► ZFS data authentication

- ★ Kontrolní součty jsou umístěny v rodičovských metadatech (Merkle stromu).
⇒ Data a kontrolní součty jsou oddělené. Detekuje/řeší poškození dat typu 1-4.

Sector checksums



ZFS data authentication

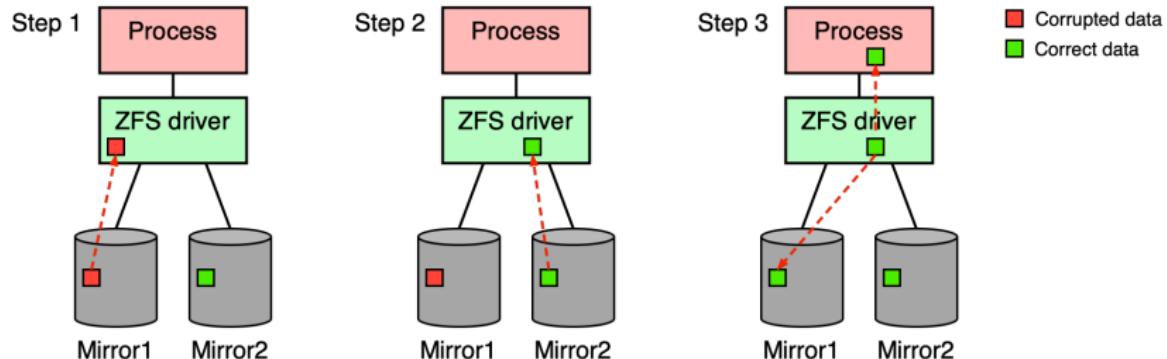


- **Automatická oprava dat (Self-healing data)**

- ▶ Díky kontrolním součtům je FS schopný sám detekovat při přístupu k datům chybu a díky redundanci dat v RAIDu ji následně i sám opravit.

- **Příklad: Detekce a oprava chyby v ZFS - RAID 1 (zrcadlení).**

- 1 Ovladač ZFS načte datový blok z prvního zrcadla a detekuje ho jako poškozený.
- 2 Ovladač ZFS načte datový blok z druhého zrcadla a detekuje ho jako správný.
- 3 Ovladač ZFS předá správná data procesu a současně opraví poškozená data na prvním zrcadle.



Použité zdroje

- ① A. S. Tanenbaum, H. Bos: *Modern Operating Systems (4th edition)*, Pearson, 2014.
- ② W. Stallings: *Operating Systems: Internals and Design Principles (9th edition)*, Pearson, 2017.
- ③ A. Silberschatz, P. B. Galvin, G. Gagne: *Operating System Concepts (9th edition)*, Wiley, 2012.
- ④ R. McDougall, J. Mauro: *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture (2nd edition)*, Prentice Hall, 2006.
- ⑤ *To FUSE or Not to FUSE: Performance of User-Space File Systems*. [Online]. Available: <https://www.usenix.org/system/files/conference/fast17/fast17-vangoor.pdf>. [Accessed:8-Mayr-2019].
- ⑥ J. Bonwick, M. Ahrens, V. Henson, M. Maybee, M. Shellenbaum: *The Zettabyte File System*. [Online]. Available: https://www.academia.edu/20291242/Zfs_overview [Accessed:8-Mayr-2019].