

1)

## Model Výpočetního systému

CLI/GUI

User

↓  
User interaction

Applications => VLC, Docker

↓  
Library calls  
API

Libraries - libc, std

ABI

User Mode

Kernel Mode

Operating System

System ISA

Firmware (bios, SMC)

System management Controller  
- Fans, battery

Hardware - CPU, RAM, NVRAM, Busses, HDD, SSD, GPU, NIC

Network Interface Controller

## Procesor - CPU

- Implementuje ISA - Soubor instrukcí, co umí provést (ADD, SUB)
- binární program lze spustit pouze když byl zkomplikován na stejném OS a ISA

Kernel Mode - vše povolené, běží v něm jídlo OS

User Mode - omezený mode, které mohou manipulovat s periferiemi:  
- běží v něm user procesy

## CISC - Complex Instruction set Computer

- Instrukce dělají komplexní operace  $ADD R_1, R_2, R_3$

$$\Rightarrow R_3 = R_1 + R_2$$

## RISC - Reduced Instruction set Computer

LOAD ...

ADD ...

STORE ...

- Program v RISC bude typicky delší než v CISC

## Zpracování Instrukcí

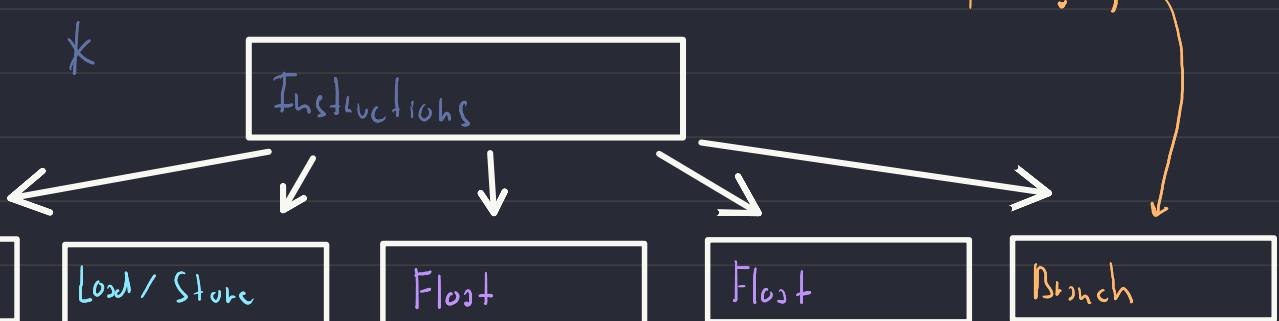
1) Fetch - načtení instrukce do pozice CPU

2) Decode - dešifrování instrukce, rozložení operátoru do registrů

3) Execute - provedení instrukce

4) Retire - uvození výsledku

Podmíněné skoky  
(BRNE - skok pokud  $!=$ )



Instruction Pipelines - instrukční proudové jednotky

## Inorder zpracování Instrukcí

- v pořadí definovaných předložených
- se využívá s paralel. horší využití CPU

ADD R1 R2 → Int pipe 1 se využívá  
ADD R3 R4 → Int pipe 1 se využívá

## Out of Order zpracování Instrukcí

- shrnut. maximizovat využití CPU
- počtu nové instrukce závislosti na předešlých výkonu se

ADD R1 R2 → Int pipe 1 Předložené  
ADD R3 R4 → Int pipe 2

Jedno jádro CPU - pouze jedno. soubor pipe viz \*

Více jádřů CPU - více soubor pipe

## Hyper Threading - funguje jen s 2 jádři

- Duplicace kritických částí - mechanismy, registry, program counter



Execution Resources - Int pipe, Flot pipe, ...

## Multi Threading

- 1 jádro je schopné zpracovávat zároveň 2 vložky
- Software concept - na úrovni OS
- vložka sdílí prostředky CPU

## Příkladů

- kreslícího nebo hrajícího uživatelského rozhraní (myš, klávesnice)
- obslužného v kernel modu

## Počítač

RAM - drží data pouze při napájení, jsou zde spuštěné aplikace a OS

NVRAM - drží data i bez napájení, obsahuje BIOS, programy → oddíl Boot

## Sběchnice

- řečou plně data mezi částmi OS (PCI-Express, ...)

## Operační Systém

- poskytuje funkci aplikacím / uživatelům

Úkoly 1) Správa a sdílení prostředků

2) Poskytuje funkci aplikacím a uživatelům

# Jednotky OS implementace

- 1) Systémové procesy, vložek
- 2) Systémová IPC - Inter-process-Communication
- 3) Systémové paměti (Allocation, Deallocation)
- 4) Systémový disk. ukládáč (SSD, HDD)
- 5) Systémové systémy souborů
- 6) Systémová In/Out politiky (pricházení, odeslání)
- 7) Systémové síťe (Sítové rozhraní, IP Stack)  
↓  
IP, TCP, UDP, ...

## Aplice - typicky běží v User mode

- komohou nejimo používat prostředky, zásoby OS
- OS kontroluje progr., řídí k prostředkům

Modely OS - MultiUser, Multiprocess, Uniflow 90% jde v C

## Přístup k paměti

UMA - více identických jader, propojeny sběrnicí

- jedník řídí paměť, řídí linky pro všechny jádra

NUMA = Non-uniform memory Access

- přístup k lokální paměti jádra je rychlejší, jinam pomalejší

2

## Spustitelný soubor programu

Obsahuje: Data, Text, kódování

- je závislý na OS a architekturě CPU

## Proces

- má složenou strukturu (vlákno, paměť, otevřené soubory, sockety)

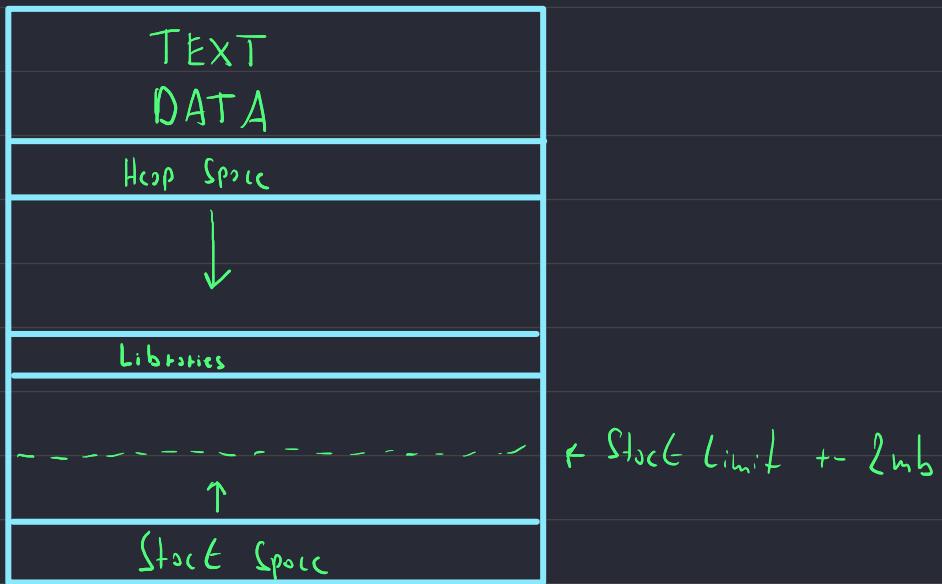
- každý má implicitní Main vlákno

- Jádro OS si může informaci PID, PPID, User/Root, page table, File Deskriptory

- při vzniku zkopírovat co má být a změnit co je potřeba



Read only topic → Copy on write



## Vlkho

- jdech proces může mit více vláčků
  - podle toho kolik má CPU jadra, kolik můžeme najednou zpracovávat vláčků
  - většinu prostředků sdílí s procesem
  - kernel má pro každý vláček kópu informaci
- => TID, Zásobník, Informace pro přepínání kontextu, plánovaní vláčků

Vytvoření procesu fork(), execve()



Adres. prostor od rodice



New Adres. prostor

↳ Text, Data

- při ukončení procesu se počítají priority čidla hodin

Výsledek ukončení  $\Rightarrow$  return - dosáhl jsem na konec

exit() - ukončení zo života

- Zobit jásdka - dílna 0, přistup mimo paměť, Signal

- vytvoření vložek  $\Rightarrow$  Posix Thread Library, M. Win. API

### Přidělování CPU

- vložku řídí přiděleno podle plánovacích kritérií (Priority)

- přidělují mu časové kvantum, po vložení odsvědčuje jásdu CPU

↑

přetížení ho časovače

### Přepínání kontextu

- mechanismus střídání vložek v používání CPU

- Kontext = informace o stavu patřícím pro běh vložky

provedení:

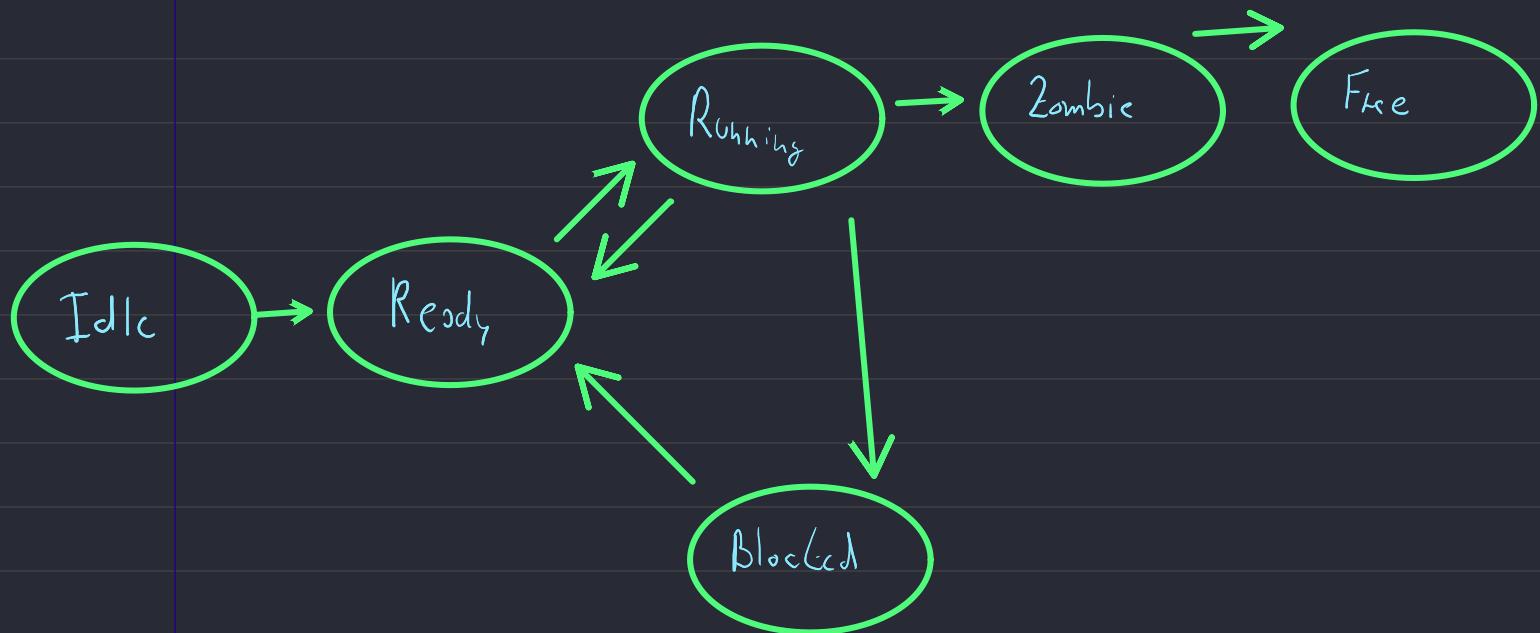
1) Vložení kontextu

2) Naplánování nového jásdu

3) Nastavení kontextu nového vložky

## Zivot Vložky

- Idle - vzhled
- Ready - čeká na CPU
- Running - běží na CPU
- Blocked - čeká na událost (input, wait(), ...)
- Zombie - vložka je ukončována
- Free - kompletně zrušeno (Terribly stav)



## Casové Závislosti chyb

- více vložek používají (čte, zapisuje) sdílenou prostředky

## Kritická sečce

- místo v kódu kde se používají sdílené prostředky

Vzájemně vyloučen!

- vložením neú dovoleno přistupovat najeďhou

Problemy při použití synchronizace

- **DeadLock** - situace, kdy nelze uloven ček na událost, kterou musíte vypolutit parci jedno z čekajících uloven

- **LiveLock** - situace, kdy nelze uloven vytahovat učitelný výpočet (má svůj stav), ale nemohou výpočet dokončit

- **Hlásorvání** - situace, kdy je uloven ve stavu Ready před bibího a dleto nemá CPU

Aktivní čekání

- Sdílený protokol indikujíc dosaženosť kritické sečce
- při vstupem do: zámčicí sečce - aktivní kontroly do když se neuděl

Odmítnutí sečce - změní hodnotu a vstoupí

- po opuštění odmítnutí

- stejně může Race Conditions =>

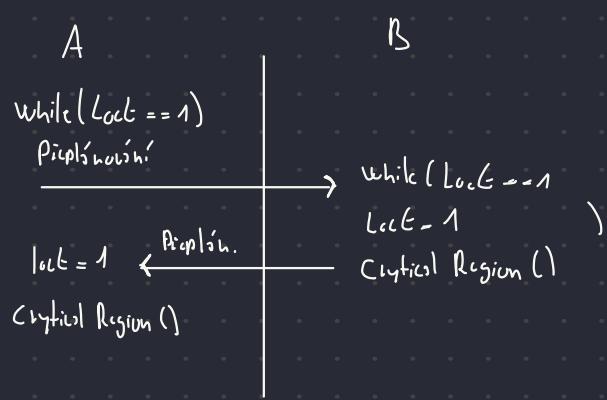
lock je n 2zášlu 0, while v A staní

daje t přeplňování, while v B staní

hostaví lock, vydje do krit. sečce

Přeplňování, A hostaví lock a vydje do

kritické sečce



## Instrukce TSL

- Hypotetická instrukce pro implementaci optimálního cílení na HW úrovní

1) Načte obsah slouž 2 paměti do registru

2) Nasloží obsah slouž do hachované hodnoty

Enter Region:

1. TSL Register, Lock	zkopíruje původní obsah lock do registru a zapiše 1 do Lock
2. cmp Register, 0	Byla původní hodnota žádoucí 0?
3. jne enter_Region	Ne? Zkus znova loop
4. tet	Aho? vstup do kritické sečce

- Nevhodný - čekající vložení zatíží CPU na 100%

## Inverzní prioritní problém

- Podminky - OS používá fixní prioritní frontu

- A má nízkou priority a je v kritické sečci

- B má vysokou priority a čeká na vstup do kritické sečce

- B bude stát pícdibít, A klidovit

## Blokující systémové volání

- implementování pomocí datových struktur, které umožňují:

1. Pamätať si stav kritické sečce (Open / Locked)

2. Udržať sčítanou vložku, kdežto čekají na vstup do kritické sečce

## Před vstupem do kritické sekce

- Změnící se sekce: Systémové volání - zablokující ho, přidá do fronty  $\Rightarrow$  Pasivní čekání
- Odčekávání sekce: Systémové volání - zapomítající si, že sekce je zavírá a vstoupí

## Po opuštění kritické sekce

- Systémové volání - pokud nebyl fronta prázdná, probudi první užívatele, pop
  - pokud je prázdná odčekávání sekci

$\Rightarrow \frac{\text{Mutex}}{\text{}}$

- metody Lock / Unlock, splňujíc definici výše
- čekání je zodolatelné
- změna stavu je užití težie

## Producent - konzument

- Producent - produkující data do sdíleného bufferu
- Konzument - konzumující data ze sdíleného bufferu

Problémy:

- 1) Sdílený buffer - mutex
- 2) buffer prázdný - zablokuje konzumenta
- 3) buffer plný - zablokuje producenta

## Podmíneký přimícha

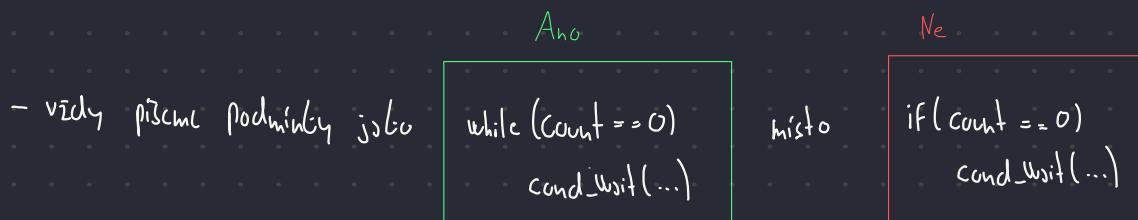
- využívá mutex

`CondWait(var, mutex)` - volá se u zavřeného mutexu

- odemčí mutex, zablokuje vloženo do funkce hudec signálizaci
- po signálizaci je mutex opět uzavřen

`CondSignal(var)`

- odemčuje akcioun jednu z vložek



- U ● by se mohlo stát, že nám jiné vložky dodačkou schovat
- potom by dalo být neplněnou po cond\_wait

## Semafory

- má celočíselný čítač
- fronta zablokovaných vložek

`Sem-Init(sem, value)` - nastaví čítač na value, vyprázdní frontu vložek

`Sem-Wait(sem)` - pokud je čítač větší než 0, sníží se  
- pokud 0 => zablokuje vložku, vloží do fronty

`Sem-Post(sem)` - pokud nejdáš vložku odebírá v frontě, jedno z nich probudí  
- pokud je přesně zvýšil čítač o 1

## Bariéry

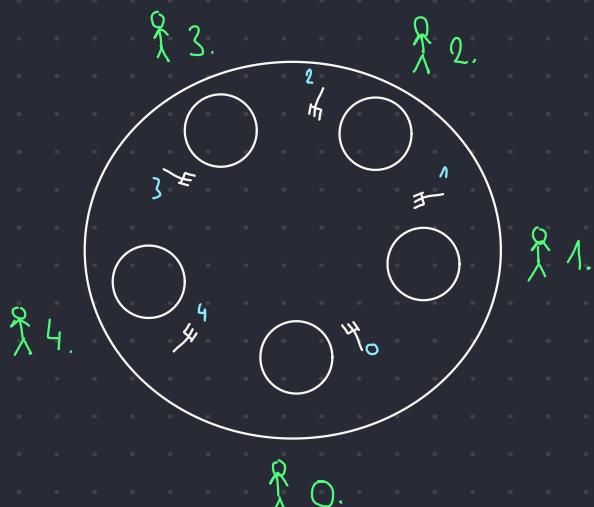
- synchronizace interakčních výpočtů
- výpočet může mít výsledky
- počítá se než value vložen do konci výpočtu, poté výsledek počítací

`Bariéry_init(bariéry, value)` - nastaví čísla na value, vypořádá funkce řídící vložení

`Bariéry_Wait( bariéry )` - pokud je číslo větší než 1, snížíme ho, zahrabujeme vložku o  
- pokud je <= 1 => přidáváme vložky vložku, nastavíme číslo na value

## Ukázka synchronizačních problémů

### 1) Vezem řidiče trolejovou



Názvy:

```
While (True)
{
    think()
    Take left()
    Take right()
    eat()
    put left()
    put right()
}
```

=> Rozložit se pokud všechny udelejí take left

=> Nicdo nesmí mít 2 náměstí jít => DeadLock

- Vylépší - pokud nemá dostupná pravá výhledka, vložíme levou a opakujeme

=> Předtím pokud budou všechny dítka to samé vezmou levou, když ji pravou, položí => LiveLock

Správné řešení - h podmínek pro každých

### Philosofci:

```
While (True):
{
    think();
    Take Farts();
    eat();
    put_Farts();
}
```

### Put Farts:

```
mutex.lock();
Farts[Left] = available;
Farts[Right] = available;
cond.signal(Left);
cond.signal(Right);
mutex.unlock;
```

### Take Farts:

```
mutex.lock();
while (!Farts_available(i))
{
    cond.wait(...);
}
Farts[Left] = used;
Farts[Right] = used;
mutex.unlock;
```

### Farts\_available:

```
return Farts[Left] = available
&& Farts[Right] = available
```

## Píšoucí a čtenáři

- databázic, píšoucí zapisují, čtenáři čtou
- více čtenářů může číst, pouze jeden píšoucí zapisuje
- hajněji píše mutex - pouze jeden čte v jednom momentu
- lze je kontrolovat v čtenářích

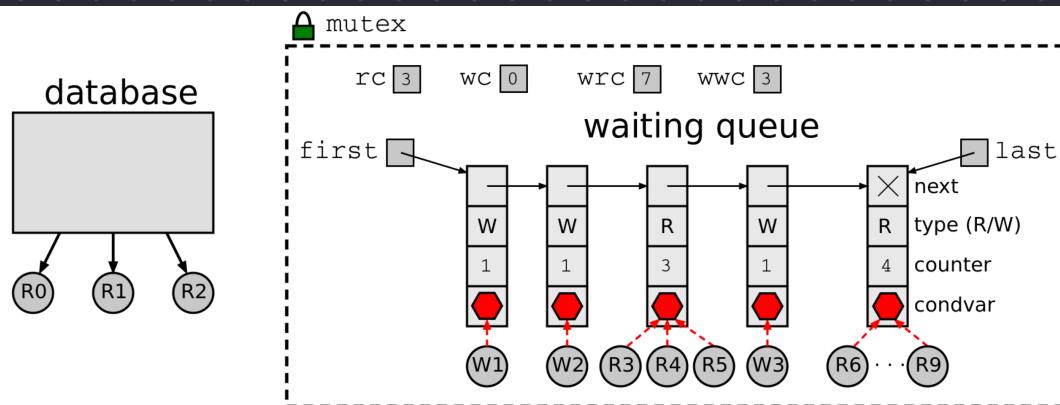
```
int      rc = 0;          /* reader counter */
mutex_t  mutex_rc;        /* access to read counter */
mutex_t  mutex_db;        /* access to database */
```

```
void reader(void)
{
    while (TRUE)
    {
        mutex_lock(&mutex_rc);
        rc = rc + 1;
        if (rc == 1) mutex_lock(&mutex_db);
        mutex_unlock(&mutex_rc);
        read_data();
        mutex_lock(&mutex_rc);
        rc = rc - 1;
        if (rc == 0) mutex_unlock(&mutex_db);
        mutex_unlock(&mutex_rc);
        use_data();
    }
}
```

```
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 }
```

Problém - píšoucí budou předcházet čtenáři

Aby byly předcházet čtenáři, musí být píšoucí ve stejném pořadí jako usínali



- 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.

```

void reader(void) {
    item_t *item;
    type_t type = reader;

    while (TRUE) {
        mutex_lock(&mutex);
        if (wc > 0 || wwc > 0) { /* writer is there => go to wait */
            wrc = wrc + 1;
            item = update_last_item(last, type); /* update the last item of the queue */
            while (item != first || wc > 0)
                cond_wait(&item->cv, &mutex);
            wrc = wrc - 1;
            update_first_item(first); /* update the first item of the queue */
        }
        rc = rc + 1;
        mutex_unlock(&mutex);

        read_data();

        mutex_lock(&mutex);
        rc = rc - 1;
        if (rc == 0) wakeup_first_item(first); /* wake up writer in the first item */
        mutex_unlock(&mutex);
        use_data();
    }
}

```

```

void writer(void) {
    item_t *item;
    type_t type = writer;

    while (TRUE) {
        prepare_data();
        mutex_lock(&mutex);
        if (rc > 0 || wc > 0 || wrc > 0 || wwc > 0) { /* anybody is there => go to wait */
            wwc = wwc + 1;
            item = update_last_item(last, type); /* update the last item of the queue */
            while (item != first || rc > 0 || wc > 0)
                cond_wait(&item->cv, &mutex);
            wwc = wwc - 1;
            update_first_item(first); /* update the first item of the queue */
        }
        wc = wc + 1;
        mutex_unlock(&mutex);

        write_data();

        mutex_lock(&mutex);
        wc = wc - 1;
        wakeup_first_item(first); /* wake up writer/readers in the first item */
        mutex_unlock(&mutex);
    }
}

```

## Spécificités

- M = nombre d'idles + actifs
- wc = nombre de clients dans la file d'attente

$\Rightarrow$  Résolu par optimisation, donc pas de deadlock (la queue n'a pas de limite)

```

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

```

```

void customer(void)
{
    mutex_lock(&mutex);
    if (wc < M)
    {
        wc = wc + 1;
        sem_post(&customers);
        mutex_unlock(&mutex);
        sem_wait(&barbers);
        have_haircut();
    }
    else {
        mutex_unlock(&mutex);
    }
}

```

```

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 }

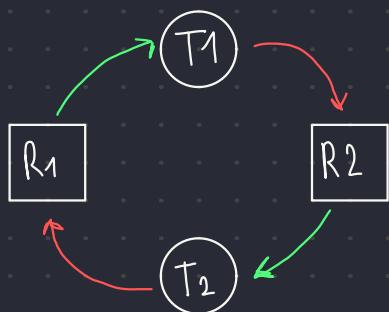
```

## Sdílené prostředky - pojít, mutex, tiskárna

- Odstranitelné (preemptable) - odložení procesu může
- Neodstranitelné (non-preemptable) - tiskárna (nemůže být oddělena běz hrozby)

## Alokaci graf

- uzel = prostředek / vlastno
- hrana - od vlastnosti prostředku = vlastnost čeká na daný prostředek
  - od prostředku k vlastnosti = vlastnost má prostředek



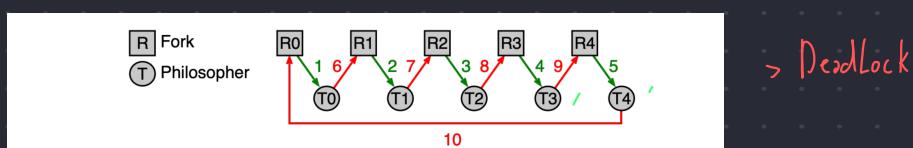
- graf obsahující cyklus => DeadLock

## Coffmanovy podmínky

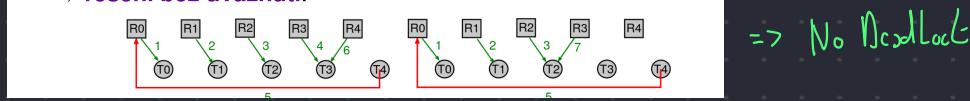
- 1) Vzájemné vyloučení - každý prostředek je přidělen pravě jednomu vlastnímu nebo jiné volné (nech' sdílen)
- 2) Neodstranitelnost - prostředek nemůže být oddělen, vlastník se ho musí sám vrátit
- 3) Drž a čekaj - vlastno, které má již přidělené prostředky může čekat o další'
- 4) Existence smyček - viz. Alokaci graf

## Příručka kruhového čekání

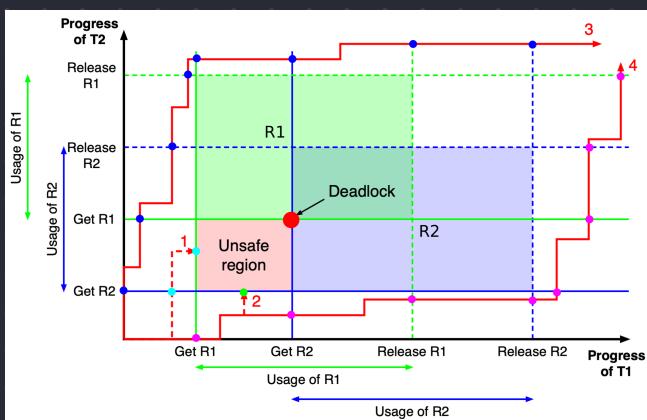
- prostředky si očislujeme a budeme vidět zákonitost ve vzrůstajícím pořadí



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



## Deadlock lze odhalit i z grafu



## Předcházení Deadlocku

E - vektor existujících prostředků

F - vektor volných prostředků

A - matice přidělujících prostředků

M - matice chybějících prostředků

Q - matice požadavků

$$M = Q - A$$

## Pořadotvory:

- zhádne všechny pořadotvory všech
- po pozití již vše uvolní

## Bezpečný stav

- stav, ve kterém pořadotvory obsahují všechny vložené
- aby byly uspořádány všechny vložené

## Bonc'iu Algoritmus

- postup zůstává v bezpečném stavu pořadující prostředky
- jinak vložku zablokuje

## Implementace procesů

- Jako OS si pamatuje info o všech procesech  $\Rightarrow$  Tabulka procesů
- každý proces = PCB (process control block) - struktura
- Počet procesů je dočasné limitován (ochrana proti Bombu)  $\Rightarrow$  limit 100

## Process Control block

- číslo procesu PID, parent PPID, SID (session ID)
- EUID - effective user id (vzájemně může spouštět procesy jenom uživatel jiný  $\Rightarrow$  Sudo)
- RUID - real user id
- GID - group id
- informace o slotových prostředcích - paměť, otvírací sculapy

## Thread Control Block

### - TID

- informace pro přepínání kontextu - registry CPU, řídicí, čítavou registry (čítací instrukce)
- typ plánuváního algoritmu, priority, stav, události w běhu číslo, využití CPU, dřívud přepnutí kontextu
- lze monitorovat - ps, top

## /Proc

- je zde pseudo filesystem - lze zde hostitovat parametry jádra → napsi /proc/sys/kernel/threshold\_max

2 způsoby implementace:

### User Level Threads

- historicky v OS kde nel. process 1 vložku
- zpracováváno jako User-level Environments
- kooperativní plánuvání - vložka počítače přichází zpět fyzické
- jádro OS nemá žádné info o vložkách, spoluji je jen jeden process

### Kernel Level Threads

- všechny moderní OS - Linux, Mac OS
- všechna správována jádrem OS
- jádro jim přiděluje CPU
- počet vložek pracuje blokující volání, jádro ho zablokuje
- virtuální, systémové volání, přepnutí kontextu  
= přepnutí z User Mode do Kernel Mode

## Plánování vložek

- mnoho strategií - vhodné pro různé aplikace (Text editor → malý odkaz, File Manager - hodně CPU - výkon)
- potížné - synchronizace, Rozložení záložek, prioritní strategie, vložka mohou hledat
- 3 typy vložek:

1) CPU - bound - threads (využívají hodně CPU, používají mnoho blokujících operací) League of Legends

2) I/O - bound - threads (CPU v čase vložek, hodně blokujících volání) Kopírování souboru

3) Real Time threads (Rychlá reakce na událost) - Hry, Skrypc :o

- vložka odvádí CPU potřeb: Skončí - volání exit(), vzdá se CPU - pthread\_yield(),  
volání blok. sys. volání => Read(), Write(), mutex, lock()
- Příklad: od časovače, I/O operace

Priemptive Scheduling - všechno dostupné CPU procesy do časového kvantumu, poté může být odstraněno

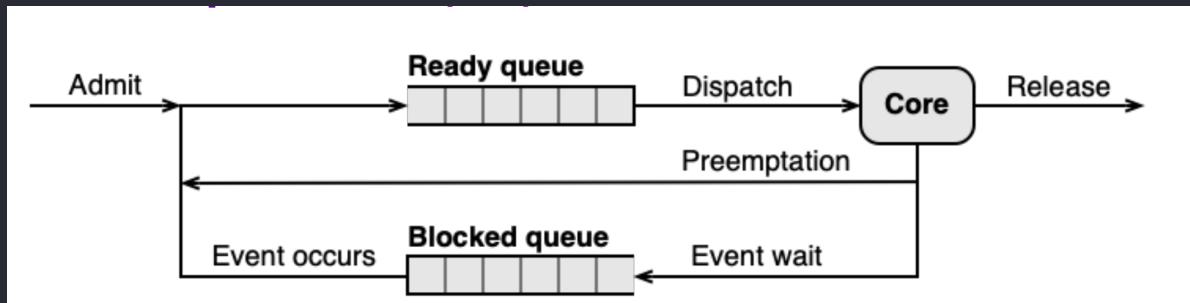
- kroužkový sdílení CPU, často její přednostní kontextu

Cooperative Scheduling

- všechno dostupné CPU a musí se ho samo vzdát
- při chybě může zabloudit CPU (povídá se proces pro svou činnost (context))
- minimalizuje se přepínání kontextu

Round-robin plánování

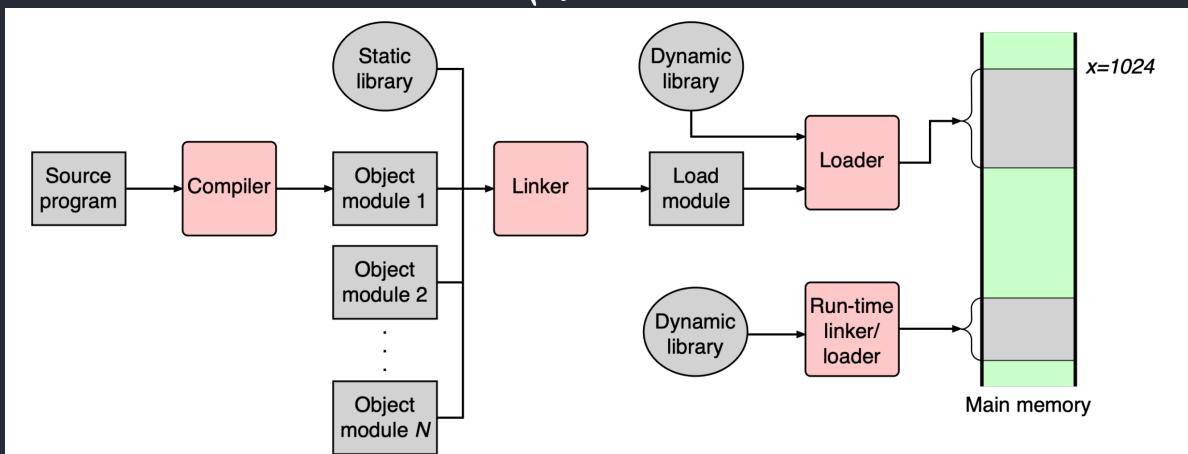
- všechny všechny přidělovanou stejnou délku časového kvantumu
- Vítězší kvantum = lepší využití CPU, větší odezva      Menší kvantum = horší využití CPU, menší odezva



- příkazy budou v jiném souboru

Příkaz k компиляции программы

- Object module - mohou obsahovat funkce a metody (nejsou spustitelné)
- Load module - spustitelný program
- Static library - archiv objektových modulů - statického sestavení
- Dynamic Library - speciální spustitelný program



## Zavdání programu do paměti:

1) Absolute loading - v binářce jsou absolutní adresy, musí být zavdán od dané fyzické adresy  
 - Linker musí dopisovat absolutní adresy při sestavování

2) Relocable loading - v binářce relativní adresy, Loadur je připraven na fyzické báhem zavdání

3) Dynamic - Runtime loading - v bin. rel. adresy, je zavdán s rel. adresami, připraveno již při přístupu

## Sestavování - Linking

- Z několika Object souborů se vytvoří jeden spustitelný = Load Module
- Symbolické adresy mezi jiné moduly musí být nahrazeny adresami vztahujícími k začátku spustitelného modulu

## Fyzické organizace paměti:

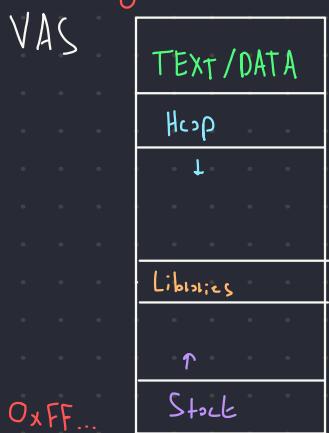
- Rychlé CPU × pomalý počítac - pomocí velkého cache (malý rychlý počítac)
- podle čeho jsou data v cache:
  - Časová lokality - bylo k nim přistupováno, bude znova
  - Přestupová lokality - budou přistupovat k datům blízko těch co aktuálně používáme

## ● 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:  $t_{avg} = t_h + (1 - r_h) \times t_m$ .

## VAS - Virtual address space

- Každý proces má vlastní VAS, mapovaný do fyzické paměti
- Struktura VAS



## Implementation VAS

1) Jako jeden logický prostor (segment) - Jeden proces v paměti

→ Identicky s fyz. pamětí

- Všechny procesy v paměti

→ Dynamické oblasti - související oblasti ve fyzické paměti

→ Struktování - množina struktur

2) Jako více logických prostorů

## Odkládání procesů - Swapping

- Pokud je nedostatek paměti oddílit se vhodní procesy do disk (např. blocked processes)

## Správa volné paměti

- OS se musí starat o volnou paměť - kde, kolik
- buď bitmapy - povídá sloučení, scénem - pomalej uvolňování, Buddy System → rychlej
- při alokaci VAS jako dynamické oblasti využívá Segmentace

## Buddy System

- žádostí o rozlohu, oblasti mají velikost mocniny 2 - rychlouší slučování

### ► Alokace paměti

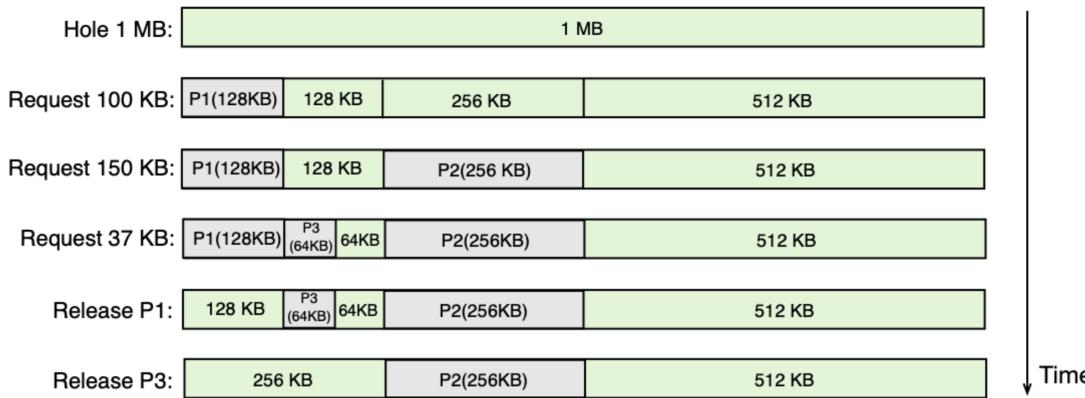
- ★ Předpokládejme, že na počátku je celá paměť volná a má velikost  $2^M$ .
- ★ Pokud nový proces požaduje paměť o velikosti  $S$ , pro kterou platí  $S = 2^M$ , potom mu přidělíme celou oblast  $2^M$ .
- ★ Jinak oblast  $2^M$  budeme rekurzivně dělit na poloviny (vždy jednu z polovin), dokud nezískáme 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.



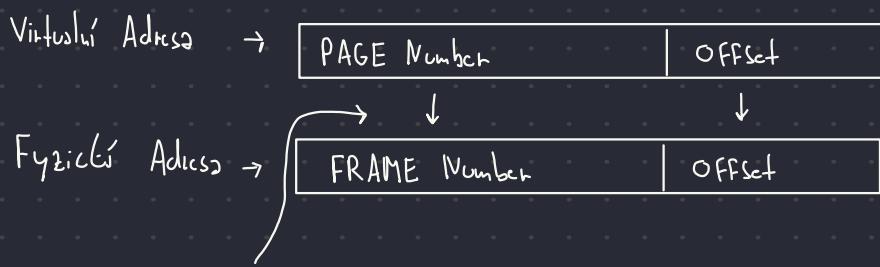
Řešení se synchronizací = Synchronizace

- Fyzická paměť se rozdělí na **Frames** stejně velikosti - 4 kB

- Logická paměť se rozdělí na **Pages** stejně velikosti - 4 kB

- OS si musí pamätať **Tabuľku Štruktúr**, **Vellosť Štruktúry**

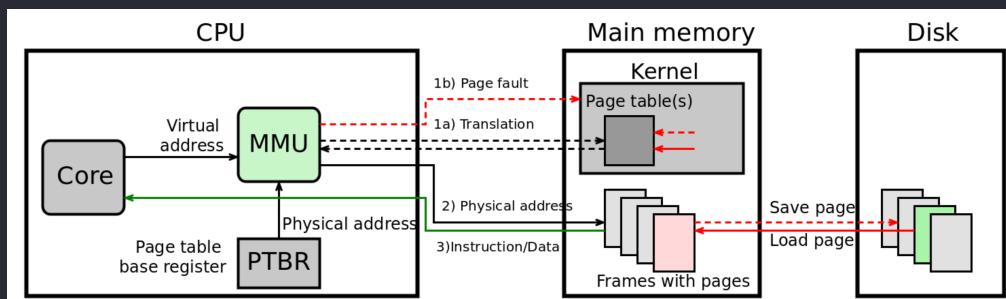
- modelný OS podporuje viac velosťí Štruktúry



Přeložka pomocí MMU

## Memory Management Unit

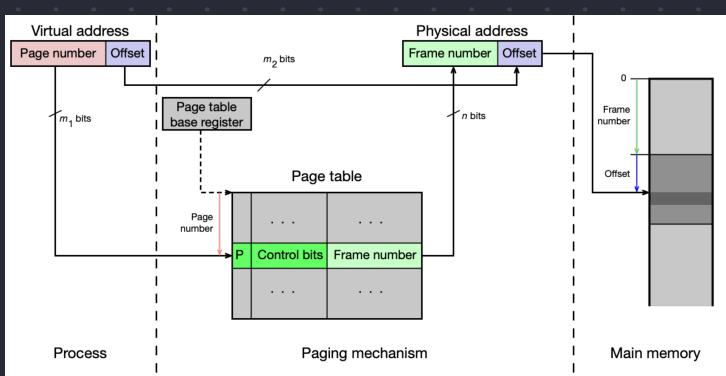
- Pokud jej potřeba stránka, MMU se stále o to snaží bylo nahrazeno v Hlavní paměti (RAM)
- Page Fault - stránka neni v paměti - je odšoupována na disk



- OS si musí pamätať, kdežto stránky jsou namapované na jednu fórmu, ktorú jeu volajú
- ⇒ Jedno/více úrovniových tabulek stránok nebo ihned všechny
- Pro využitie přehledu se používá TLB

- Tabuľka obsahuje pre každu stránku VAS daného procesu jednu řádku, ve ktorej sú uložené nasledujúce informacie
  - číslo rámca, do ktorého sa tato stránka namapovala,
  - kontrolné bity
    - ★ Present bit (P): informácia, zda stránka je v hlavnej pamäti,
    - ★ Reference bit (R): informácia, zda sa k stránke prístupovalo,
    - ★ Modify bit (M): informácia, zda bol obsah stránky modifikovaný,
    - ★ Prístupové práva
    - ★ Cache disable/enable: zda sú registre periferií mapovaný priamo do pamäti,
    - ★ Read/write (R/W): informácia, zda je možné stránku modifikovať,
    - ★ User/supervisor (U/S): informácia, zda je možné na stránku prístupovať v užívateľskom režime, ...

- číslo stránky = index do tabuľky
- jeden proces = jedna tabuľka



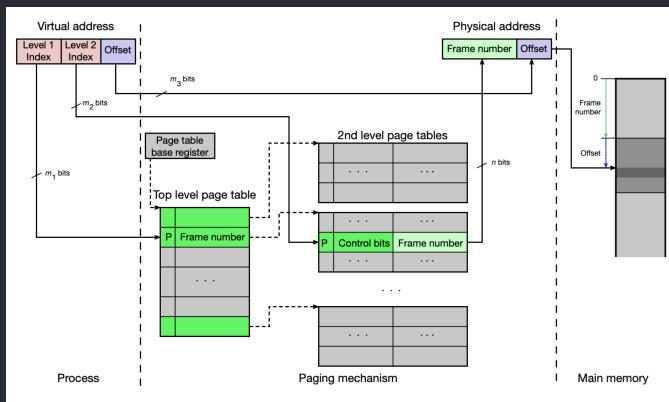
# Problém

- VAS - používá se pouze zlomky, ale potichujeme celou tabulkou  $\Rightarrow$  Příkladního prostoru!
- Řešení  $\rightarrow$  Víceúrovňová tabulka

## Víceúrovňová Tabulka

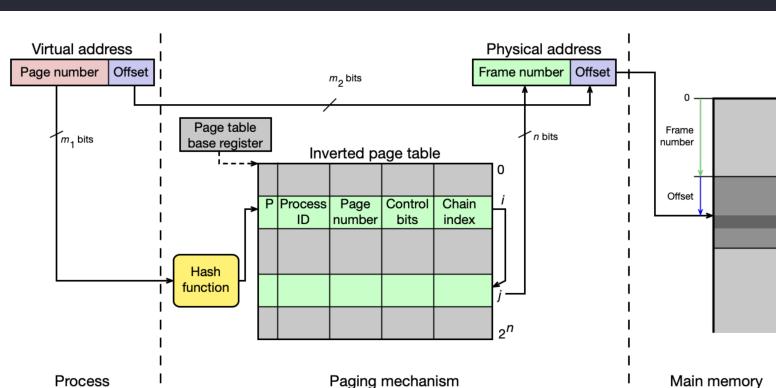
### • Pro $n$ úrovňovou tabulku stránek platí

- ▶ 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 oblasti VAS, která tyto tabulky pomáhají adresovat  $\Rightarrow$  šetří se fyzická paměť.
  - Za ušetřené místo však platí pomalejší překladem  
 $\Rightarrow$  pro urychlení překladu se používá společně s TLB.



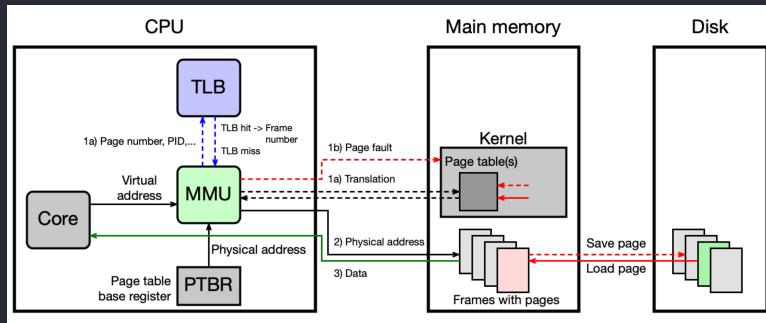
## Inverzní Tabulka Stránek

- Tabulka obsahuje pro každý rámec fyzické paměti jeden řádku, ve kterém jsou uloženy 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 hashovací funkce přepočte na index do tabulky. Protože počet stránek je větší než počet rámů fyzické paměti, několik různých stránek se může namapovat na stejnou řádku v tabulce  $\Rightarrow$  protože používá techniku zřetězení.

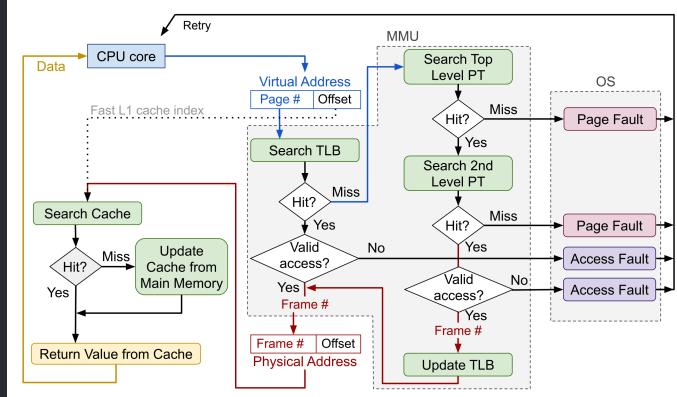


## Translation lookaside buffer - TLB

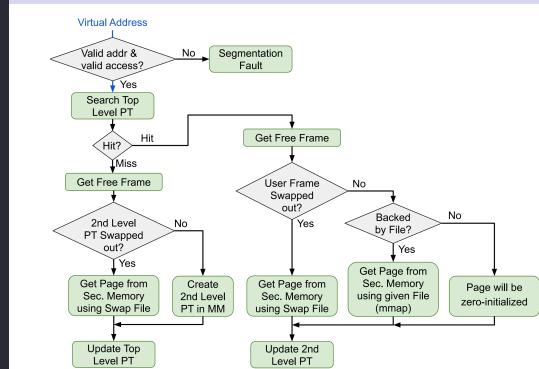
- cache - obsahuje informace o nápoledy překladačích adresách
- položka obsahuje: valid bit, číslo stránky, číslo řámečku, ASID (address space), control bits



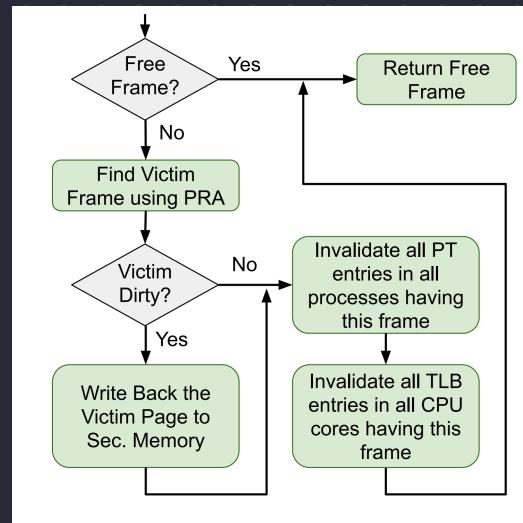
### Překlad virt. adresy pomocí TLB a 2-úrovňových str. tabulek



### Page fault: OS handler



- OS nejdřív ověřuje platnost virtuální adresy (zda neleží mimo rozsah VAS) a oprávněnost přístupu (pokus o zápis do read-only stránky<sup>1</sup>).
- Fault Exception Handler musí zjistit, na které úrovni stránkovacích tabulek došlo k výpadku.
- V každém případě OS musí získat **volný rámeček v hlavní paměti**: Get Free Frame (viz další slide a příští přednášku).
- Ten se použije pro nahrání stránky. Data mohou pocházet ze:
  - ▶ **zdrojového souboru** (například při zavádění procesu do hlavní paměti čtením dat ze spustitelného souboru, nebo ze souboru namapovaného do paměti pomocí mmap) - viz slide 32,
  - ▶ **swapovacího souboru** (data byla v minulosti odložena z hlavní paměti na disk),
  - ▶ nebo  **inicializována nulou** v ostatních případech (neexistuje odkaz na soubor: heap, stack, anon) - viz slide 32.



- Pokud hlavní paměť není plně obsazena, vrátí se libovolný volný rámeček.
- V opačném případě musíme najít rámeček, který se odloží na disk (viz příští přednáška) a tím se uvolní.
- Jenomže musíme zneplatnit překlady všech stránek, které se mapují na tento rámeček.
- Několik TLB na různých CPU jádřech může obsahovat překlad na tento rámeček a navíc takový rámeček může být sdílen více procesy.
- Důsledek: Musí se identifikovat všechny procesy odkazující na tento rámeček a zneplatnit jím odpovídající položku ve stránkovací tabulce a navíc se musí zneplatnit všechny položky všech TLB s tímto rámečkem.

# Algoritmy pro hledání stránek

- hledání počítat je plno  $\Rightarrow$  každý rámec se musí uvolnit
- musíme vybrat takový rámec, který může v blízké době potřeba

## Optimální algoritmus

- **Princip**
  - ▶ Nahradí se stránka, která má čas příštího přístupu nejdéle (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  
 $\Rightarrow$  ale lze použít pro porovnání kvality reálných algoritmů.
- **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?

Cílové stránky	2	3	2	1	5	2	4	2	3	2	5	2
Rámec	a	2	2	1	5	2	4	2	3	2	5	2
b		3										
c				1	5		5		5			

- ★ Přístup bez výpadku stránky/přístup s výpadkem stránky.
- ★ Pokud je více možností, volíme rámec 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 hardwarém 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 resetoval 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 ze neprázdné třídy s nejnižším číslem.

## FIFO algoritmus (First-In First-Out)

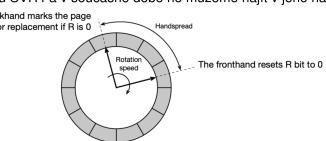
- **Princip**
  - ▶ OS si udržuje seznam všech stránek, které se aktuálně nacházejí v hlavní paměti.
  - ▶ V okamžiku, kdy se stránka nahraje do hlavní paměti, přidá se její záznam na konec seznamu.
  - ▶ FIFO algoritmus vybere první stránku ze seznamu jako vhodného kandidáta pro nahradu.
- **Vlastnosti**
  - ▶ 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  
 $\Rightarrow$  způsobuje relativně velký počet výpadků stránek.

## Clock algoritmus

- **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 ručička 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 rovnným 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.

## 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.
  - ▶ Rozvedení ručiček (handspread) společně s rychlosťí definuje časové okno, na základě kterého poznáme, zda se ke stránce nedávno přistupovalo.

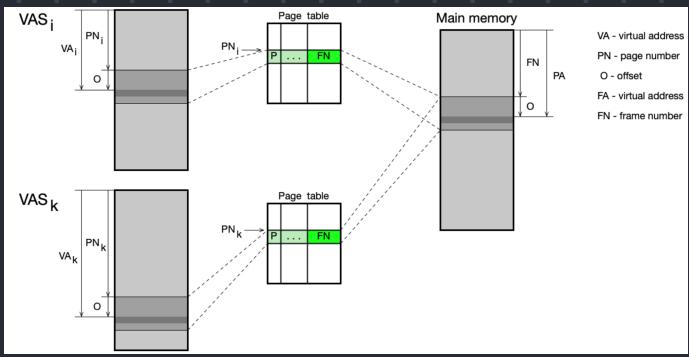
## LRU algoritmus (Least Recently Used)

- **Princip**
  - ▶ Vhodným kandidátem pro nahradu je stránka, ke které se nepřistupovalo po nejdelší době.
- **Vlastnosti**
  - ▶ Dobrá approximace optimálního algoritmu.
  - ▶ Problematiká implementace.
    - ★ Při každém přístupu ke stránce je nutné si zapamatovat informaci o "čase" přístupu.
    - ★ Vhodným kandidátem pro nahradu 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 nahradu je stránka s nejmenší hodnotou "time-of-used".

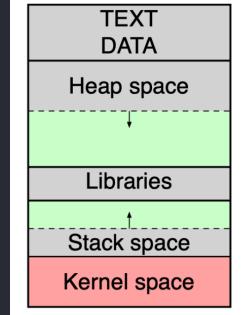
## Aging algoritmus

- Softwareová 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 nejvýznamějšího 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 presný 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.

## Sdílení paměti mezi procesy - při stránkování



- Mapování Kernel Address Space
  - 1) U stávých OS je lze VAS →
    - při přechodu z USER do kernel módu
    - potřebovalo třeba mít oddělený prostor
  - 2) U většiny nových je oddělený



Nahlášení stránek do paměti:

- ho poslat
- Paging - prostorová / časová lokality → Minimizace přenosů z disku

Co dělá OS při nedostatku paměti

- 1) Paging-out - Strategy - uvolňování stránek a uložení na disk
- 2) Swapping-out - Strategy - odložení procesu na disk

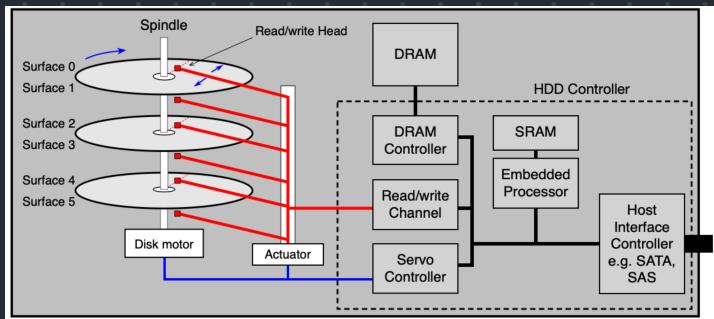
Dostoucí úložiště

- dlouhodobé úložiště informací
- SSD, HDD, RAID
- Skládá se ze sekční, dělí se na partitions

## Hard Disk Drive

- Mechanické části
  - kolo s plotenem, co má již 2 pouzdro, otevírá se stejnou rychlosťí
  - polohyblivé hlavice → čtení / zápis, jsou všudy všechny stejné dočekávání

Elektronické části - řídí disku (Disk Controller) - procesor, paměť s firmware, řídí tot dat u disk  
- výkonová paměť - buffer pro čtení / zapisování dat



detache a optus chyb

1

Sektor - nejmenší sdílenovatelná jednotka - obsahuje: Servis info pro řadič, data, ECC - error correction code

Cylinder - množina všech stropů o daném poloměru ve všech poutích

Adlesovský sektoru: CHS - state' [1,2,3]  
cylinder ↑ ↑ ↗ sector  
pouch

LBA - hole, od 0..., 2ocík sc od vnejšího okraj cylindru

## Zone-bit Recording

- Stopy dílčny do zón, aby byly konstantní počet sčítanou na stupně
  - Vnější vlny stop hůř vnitřní { }.

Pripojení disků k PC

- SATA - 1m max, 1.5-6 Gb/s
  - Thunderbolt - PCI-Express + Display port, 40Gb/s
  - SAS - 10m, 22.5 Gb/s
  - FC (Fiber Channel) - 12Gb/s, 10KM

## Rychlosť čtuní / zápis

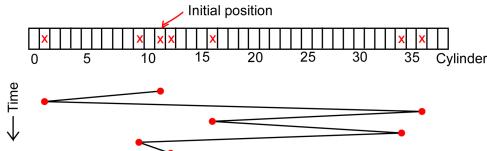
- Sect Time - doba wystawu hlawic (1-10 ms)
  - Rotational delay - hodit spletneho setravu v stupci
    - priwetne cis 1ofaciu /2

# Algoritmy přístupu k disku

## First-In-First-OUT (FIFO)

- Požadavky (čtení/zápis) jsou řazeny do fronty.
- Požadavky budou obsluženy v pořadí v jakém příšly.
- Výhody:** spravedlnost.
- Nevýhoda:** horší výkon.

Initial position: 11.  
Order of input requests: 1, 36, 16, 34, 9, 12.

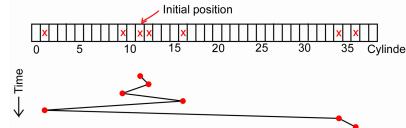


## Shortest Service Time First (SSTF)

- Požadavky (čtení/zápis) jsou řazeny do fronty.
- Nejdříve jsou obsluženy požadavky, které vyžadují nejmenší pohyb hlaviček z aktuální pozice.
- Výhoda:** lepší výkon než u FIFO.
- Nevýhoda:**

  - Hlavičky mají tendenci setrvávat uprostřed disku.
  - Vzniká problém strárnutí u požadavků z krajních pozic.

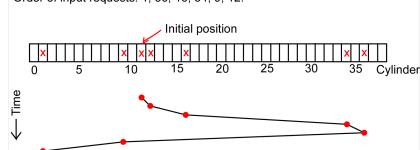
Initial position: 11.  
Order of input requests: 1, 36, 16, 34, 9, 12.



## 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.
- Výhoda:** částečně se omezil problém strárnutí požadavků.
- Nevýhoda**
  - Trochu horší výkon než SSTF algoritmus.
  - Nereší strárnutí při velkém počtu požadavků v úzké oblasti cylindrů.

Initial position: 11.  
Order of input requests: 1, 36, 16, 34, 9, 12.

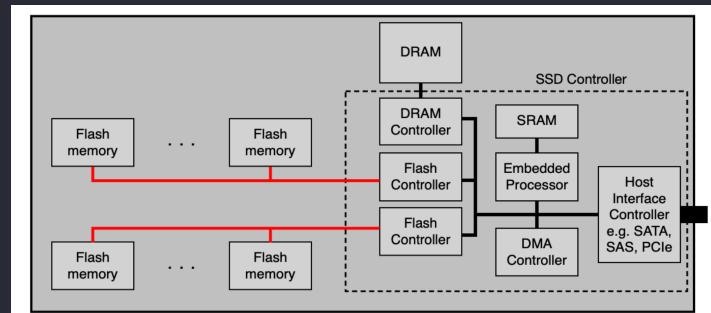


## N-step SCAN

- 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 obsluženy 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.
- Výhoda:**
  - 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

- Obsahuje žádné mechanické části
- rychlost, menší spotřeba, menší hmotnost
- menší kapacita, vysoký cena



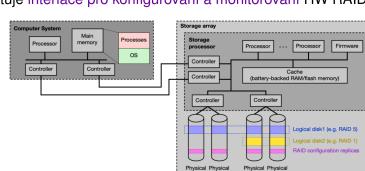
## Redundant Array of Independent Disks - RAID

- Spojí více disků pro větší kapacitu, spolehlivosť, rychlosť



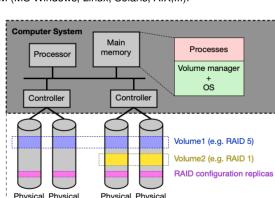
## Hardwarový RAID

- 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 firmwarém,
  - 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 jsou představují konkrétní typ RAIDu a poskytuje k nim interfejs pro OS/aplikace, které běží na připojeném výpočetním systému (OS přímo nevidí fyzické disky HW RAIDu).
  - Poskytuje interface pro konfigurování a monitorování HW RAIDu.



## Softwarový RAID

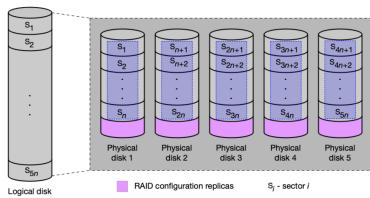
- Fyzické disky HDD/SSD (disk1, ..., disk4) jsou k systému připojeny přes příslušné sběrnice a jsou standardně spravované prostřednictvím OS.
- Volume manager (VM)**
  - Software, který ukládá (mapuje) data na jednotlivé fyzické disky a provádí nutné výpočty (např. výpočet parity,...).
  - Spravuje logické disky (volumes), které představují konkrétní typ RAIDu a poskytuje interfejs 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,...).



# Typy RAID

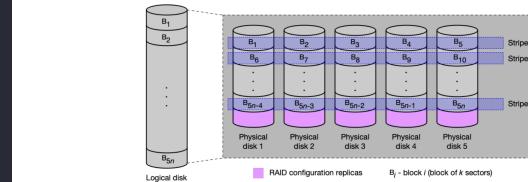
## RAID 0 – zřetězení (concatenation)

- 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.



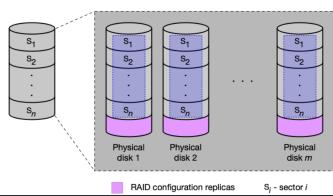
## RAID 0 – prokládání (striping)

- **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.



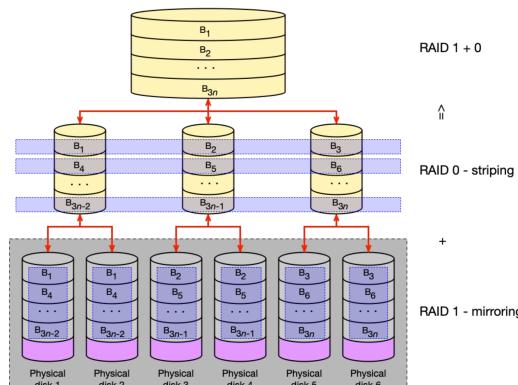
## RAID 1 – zrcadlení (mirroring)

- **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  $m$  kopie 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.



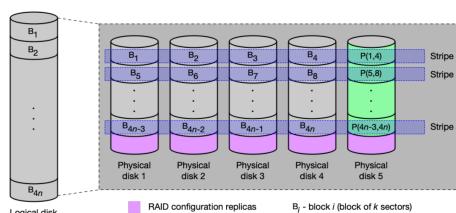
## RAID 1+0 (stripe), RAID 10

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



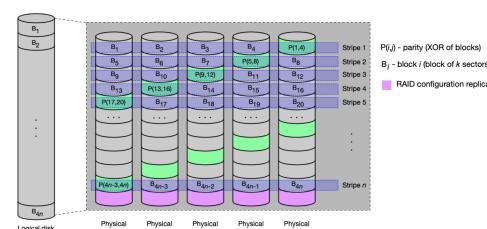
## Raid 2, 3, 4

- 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.



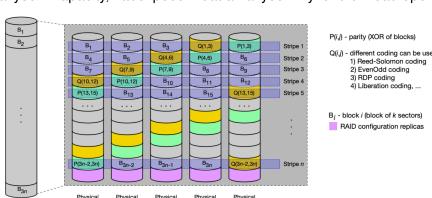
## Raid 5 – prokládání s distribuovanou paritou

- **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  $(m - 2)$  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.



## Typy disků

Directly Attached

Network Attached

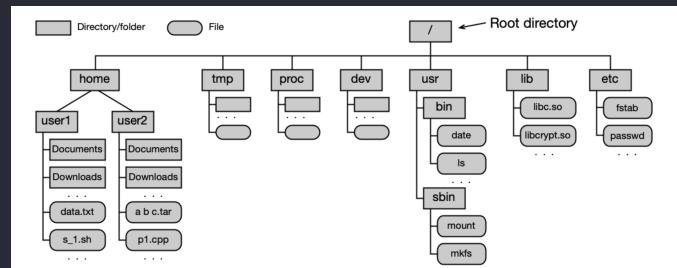
Storage Area Network

# Systémové soubory

- Uživatel ho vidí jenou složkovou strukturu adresářů →
- Soubor - > mnoho, atributy (typ, vlastník, práva, časy), data
- Na disku je:
  - Label - partition table, bootloader
  - MBR, EFI, GPT

Partitions - jednotlivé FS

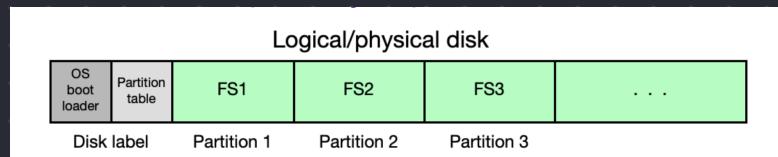
- Operace nad diskem →



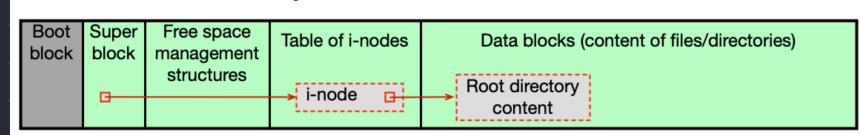
## Typické operace nad diskem a FS

- 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 na disk →



- Rozložení dat v File systému →



- Bootloader - po spuštění PC se máte Bios, ten si máte info o partitionech, dle nížm ho výběr oddílu bootovat
  - např. Dual boot Linux P1 / Windows P2

boot block co máte linux

boot block co máte windows

- Super Block - Typ FS, velikost dat. bloků, volná/slobodní bloky

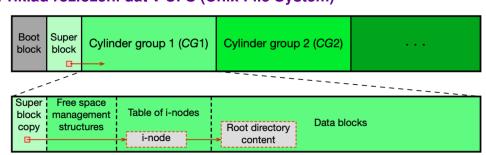
- FSMS - datové struktury pro správu volného prostoru

- Dataný blok - velikost struktury je malá protože se píše s celými bloky

- lze upravit délku všechny disku

- příklad UFS →

## Příklad rozložení dat v UFS (Unix File System)



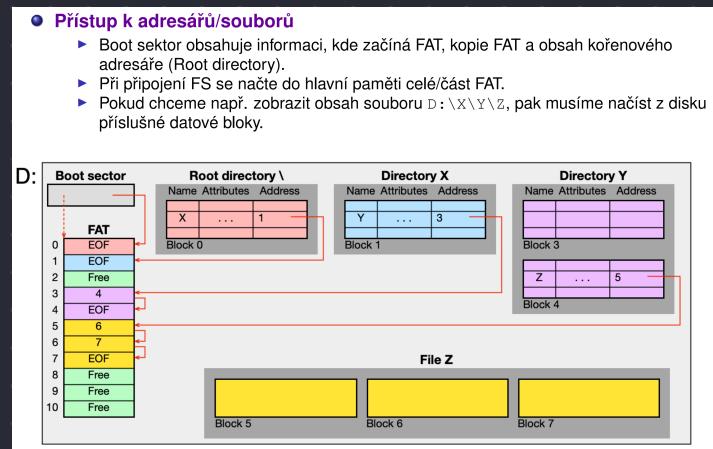
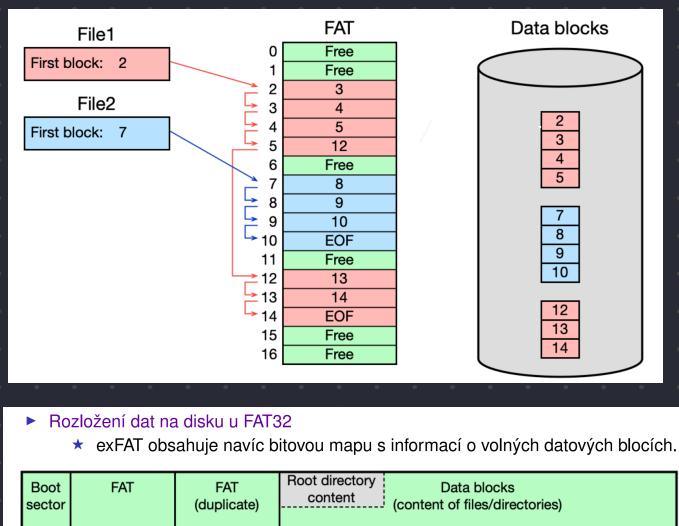
- UFS je reálný FS používaný např. v Solarisu nebo BSD.
- Z důvodu výkonu a zabezpečení dat proti ztrátě je diskový prostor UFS rozdělen do několika stejně velkých oblastí CG<sub>i</sub> (Cylinder groups), které jsou reprezentovány souvislou množinou cylindrů na disku.
- Soubor/adresář je vždy alokovan v rámci konkrétní CG<sub>i</sub> ⇒ lepší výkon (hlavicky HDD se pohybují pouze v rámci CG<sub>i</sub>).
- Pokud dojde k poškození začátku disku (např. administrátor přepíše omylem Super blok a několik prvních sektorů z CG<sub>i</sub>), pak se ztrátí data z CG<sub>i</sub>, ale data z ostatních CG<sub>j</sub> se podají většinou zachránit pokud víme s jakými parametry byl UFS vytvořen.

# Typy Allokace

- 1) Součísle oblasti - rychlé, malý overhead, problém segmentace → NTFS
- 2) Po bločích - pomalý sekvenciální přístup, hran segmentace, větší overhead → UFS, Fat 32, EXT 2, 3, 4
  - pro bloční informaci o umístění bloků → FAT / i-node

## File Allocation Table - FAT

- Tabulka má sloužit k zápisu, kde je datových bloků → zabírá hodnou paměti
- Pro každý soubor si pamatujeme adresu prvního bloku, potom žádáče
- Pro malé FS lze mít celou FAT do RAM → ulevlení, pro velké nelze

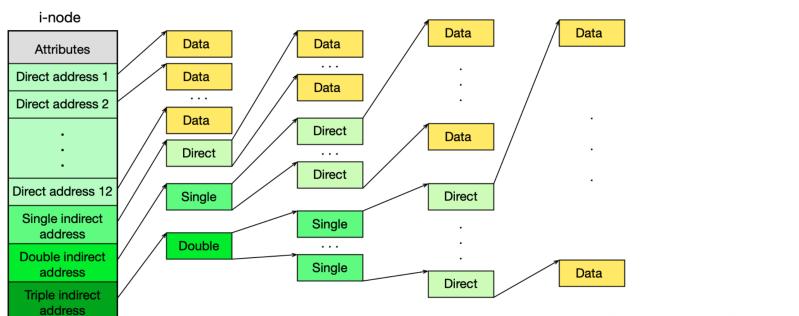


## I-node (Index Node)

- Soubor si pamatuje adresu svého I-Node
- Fixní velikost, drží pouze oděsky

### 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.

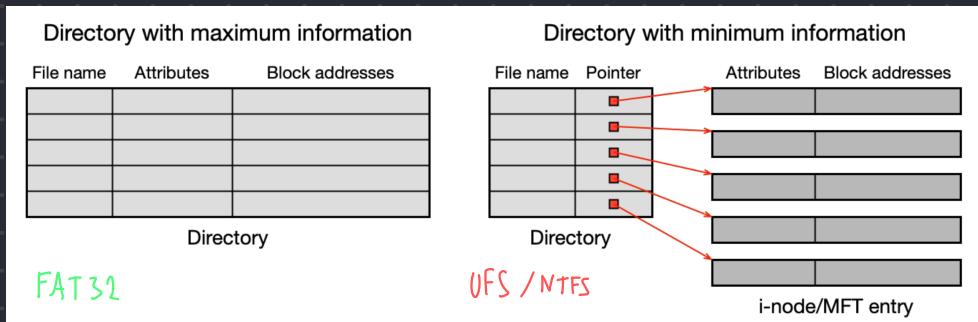


### Vlastnosti i-nodů

- ▶ 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 náhodnému přístupu je **pomalejší přístup k datům při prvním přístupu**. Při následujících přístupech se již využívá skryté 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, ...)

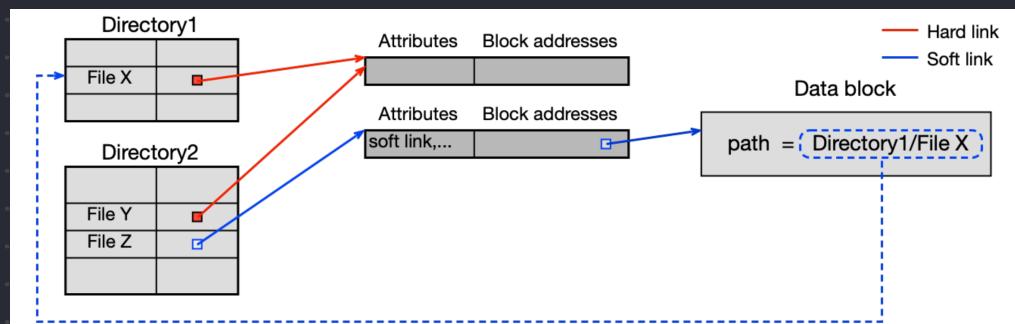
## Implementace Adresářů

- obsah je uložen v jednom nebo více datových blokách
- 2 přístupy → Adresář obsahující Minimum informací / Maximum informací



## Sdílení souborů

- Soft Link - odkaz na existující soubor / Adresář implementovaný jako cesta k souboru
- Hard Link - implementovaný jako odkaz do I-Node



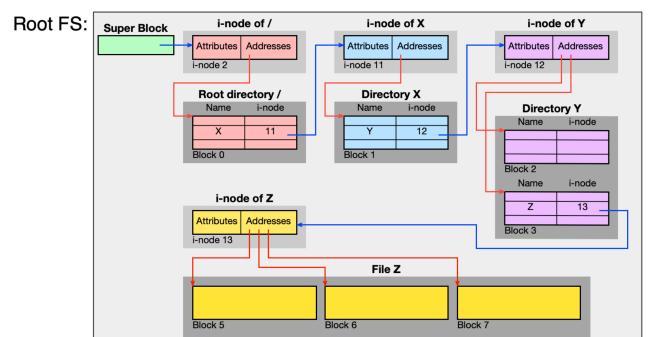
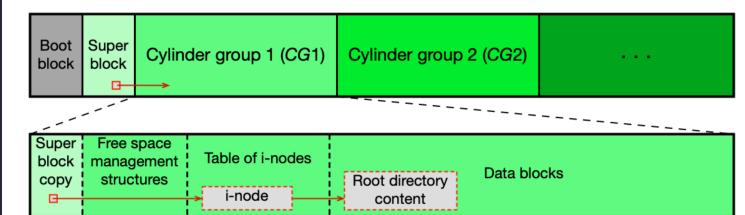
## UFS - Unix File system

- BSD, Solaris
- datový blok 8kB
- I-Node - 128B, přístup plánu, vlastník, 15 32-bit adres (12 průměrných, 3 keřícné plně, dlečí a třetí všechny)

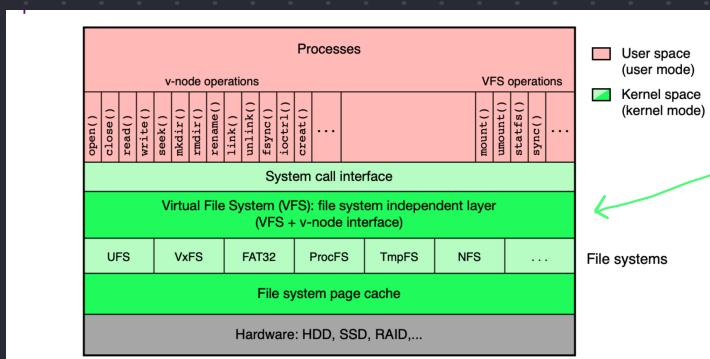
### Přístup k souborům/adresářům

- 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.

### Rozložení dat na disku

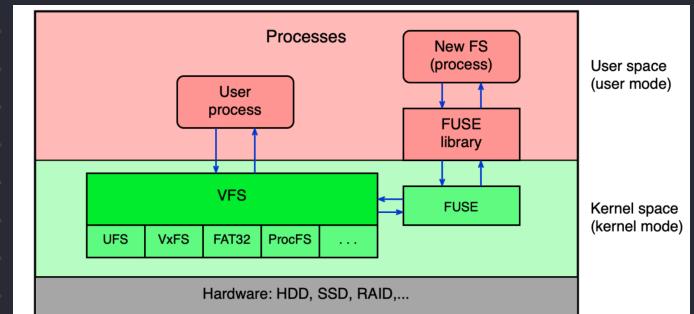


Abychom mohli používat ičiúci FS je implementačný abstrakt → VFS



## Fuse - File System in User Space

- neplníciasväti vizitáku mohou vytvoriť svoj vlastný FS
- k novému FS musíme definovať standartné operacie
- FS môže byť napiestod vzdialosť (vo súť)



## Efektivita operacii

- plasťová a časová lokalita
- Block Cache →

### Block cache/Page cache

- Velikosť datových blokù FS je obvykle navrhovaná tak, aby bola **násobkom** velikosti stránok ve virtuální stránkovanej pamäti.
- Block cache/Page cache je **množina** nedávno používaných datových blokù z FS (**obsah súboru/adresáru**) uložená v hlavnej pamäti z dôvodu zlepšenia výkonu FS.
- OS si udržuje informaci o všetkých blocích načtených do hlavnej pamäti.
- Když proces **čte obsah** adresáre/souboru, OS hľadá bloky (obsah) nejdříve v hlavnej pamäti, pokud deje nejsou, pak je načte z FS.
- Když proces **modifikuje obsah** adresáre/souboru, pak existují dva prieťupy
  - ★ **write-behind cache:** modifikace je zapsána ihned do datového bloku v hlavnej pamäti a se zpožděním do FS (běžně používané u FS na interních datových úložištích),
  - ★ **write-through cache:** modifikace je současně zapsána do hlavnej pamäti i do FS (běžně používané u FS na přenosných datových úložištích typu flash-disk).

- v Solarisu →

### DNLC (Directory Name Look-up Cache)

- Skrytá pamäť v OS Solaris, ktorá obsahuje jmena nedávno používaných súboru/adresáru a jejich v-nody (datova struktura VFS).
- Při následujícím prístupe k souboru/adresári se použije informace z DNLC, nikoliv z datových blokù FS.

## Ochranu dat pri pôdu systému / výpadku napájania

- Vytvorenie/modifikace souboru/adresáre se obvykle skladá z několika kroků.
  - ★ Modifikace struktury související s volným prostorem (i-nody, datové bloky,...).
  - ★ Modifikace obsahu adresáre, ve kterém se soubor/adresář má vytvořit.
  - ★ Modifikace metadat (položky adresáre, 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.

✓ Řešení

### Žurnálovaný FS (Journaling FS)

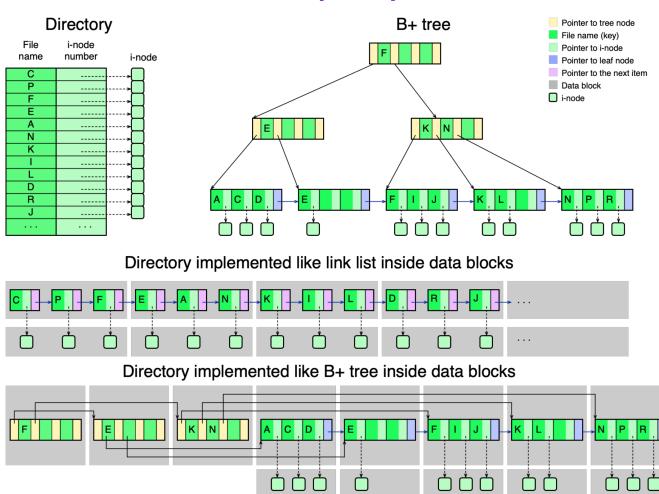
- Snaží se **ochrániť** FS pred nekonzistenčí a ztrátou dat.
- FS si alokuje na disku **speciálni oblasť "journal"**, do ktoré si v predstihu zapíše **záznam o změnách**, ktoré sa budou následne provádēt. Potom, co sa změny úspěšne zapísí do FS, je záznam z "journalu" odstraněn.
- V případě havarie, po zotavení systému se "prehraje" záznamy z "journalu" a FS vrátí do konzistentního stavu.
- "Journaling" je implementován ve většině současných FS (např. UFS, EXT2/3/4, NTFS,...).
- Z dôvodu výkonu se do "journalu" většinou **ukládají pouze metadata** ale nikoliv obsah datových blokù.
- "Journal" s nemusí **vždy nachádzať na stejném disku** ako samotný FS, ale může byt umístěn např. na rychlejším SSD.

# Moderní FS → BTRFS (B-Tree FS), APFS (Apple FS)

## Mezi jejich důležité vlastnosti patří

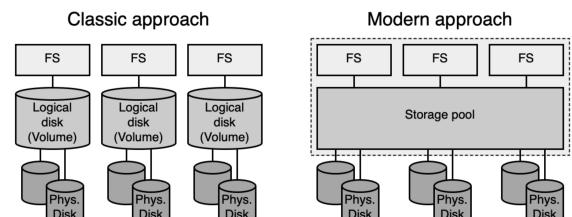
- Při implementaci FS se používají B stromy/B+ stromy.
- Většinou reprezentují kombinaci SW RAIDu a FS.
- Integrita dat (FS je neustálé v konzistentním stavu).
  - ★ Redundance pomocí RAID ⇒ ochrana proti výpadkům HW komponent.
  - ★ Copy-on-write transactional object model ⇒ ochrana dat proti výpadku napájení,....
  - ★ Kontrolní součty dat/metadata ⇒ detekce a oprava dat/metadata.
- Podpora efektivního vytváření "clonů" a "snapshotů".
- Podpora šifrování dat.
- Podpora komprese dat a deduplikace datových bloků.
- Jednoduší administrace FS (obvykle se vystačí pouze s několika příkazy, např. zpool a zfs v ZFS).

## Příklad: Použití B+ stromů při implementaci adresářů



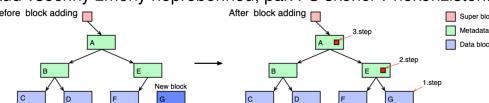
## Kombinace SW RAIDu a FS

- 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.
    - ⇒ Jednoduší administrace (zvětšování/zmenšování jednotlivých FS).

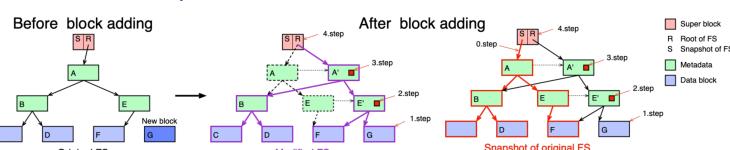


## Integrita dat

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



Siy jdu koutat na Hotej

## 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 dat).
- 2 Lost writes: Disková operace "write" se neprovedla, 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/opravuje pouze poškození dat typu 1.
- ZFS data authentication
  - ★ Kontrolní součty jsou umístěny v rodicovských metadatech (Merkle strom).
    - ⇒ Data a kontrolní součty jsou oddělené. Detekuje/řeší poškození dat typu 1-4.

