- **C a C++**

false = 0
true ≠ 0

char ≤ short ≤ int ≤ long ≤ long long < float < double

→ implicitní konverze vždy do silnějšího typu ⇒ int · long = long

- **array** int a[4] ← random hodnoty → adresa pole = bázová adresa
  int p[] = {1, 2, 3}
- **string** = array charů končící na NUL = \0 sizeof("ahoj") = 5
- **pointer** → je to číslo (adresa), ale musím říct, že to je pointer na něco

```
int v = 8;
int* pv = &v;     & mi dá adresu
*pv = 4;          *pointer = přístup na tu adresu, nyní v = 4
```

- **reference** → ukazatel, co ale ukazuje jen na 1 fixní adresu → jen v C++

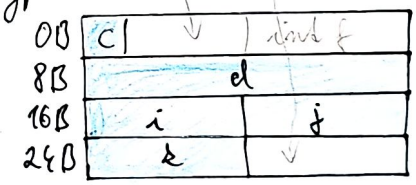
```
int v = 8;
int &pv = v;      pv je nyní jen jiné jméno pro v
pv = 4;           → v = 4
```

- **struct** data {

```
char c;     1B
double d;   8B
int i, j, k;  4B
```

};

→ vnitřní a vnější zarovnání alignment
→ typ velikosti X musí být na adrese A: A%X = 0



→ vnější zarovnání na velikost největšího typu v té struktuře
⇒ aby se to dalo dát do pole

```
void funkce(data in, data *out) {
    out->c = in.c;
    out->d = in.d;
}
```

→ v C je vše předáváno hodnotou
→ předání referencí pomocí pointru
→ přístup k membrům reference
(*ptr).field ≡ ptr->field

C++ reference
data & out
out.c = in.c

- **konstanty**

```
const int jeden = 1;          ← neměnná proměná
constexpr int druhej = 2;     ← compile-time konstanta - není v paměti ⇒ lepší
```

- **preprocessing** - tohle se řeší na úplném začátku kompilace

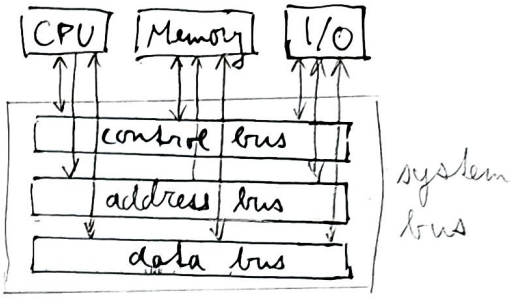
```
#include <stdio.h>   → prostě se tam nakopíruje obsah toho souboru
#define N 100
#ifdef ...
#endif
```
} ugh

• CPU architektura

• Von Neumann

- program a data ve stejné operační paměti
- 1 sdílená sběrnice ⇒ vždy jen 1 transakce
  ↳ když CPU píše do paměti, tak nemůžeme z paměti psát na monitor

pomalý



• Harvard

⇒ trochu rychlejší

- oddělená data a code memory → sběrnice z CPU i do code mem
- pořád v nějakých μC ↳ je potřeba více adresových prostorů procesoru

• Reálná

- CPU má paměťové řadiče a paměti jsou připojené vlastní sběrnicí přímo k CPU
  ↳ dokonce má víc kanálů do 1 paměti ⇒ paralelizace
- grafika je buď uvnitř procesoru nebo externí grafická karta připojená přes PCI express
- South Bridge
  - připojený sběrnicí k CPU
  - jsou na něj připojené všechny periferie
    ⇒ HDD, DVD drive, myš, síťová karta, zvuková karta
- dřív byl i North Bridge pro paměť → pomalé → teď to dělá procesor
- všechny sběrnice jsou peer-to-peer = mezi 2 zařízeními (dřív sdílená)

• Co je ta architektura?

• Instrukční sada - ISA

↳ x86, MIPS, ARM

- specifikace jak se ten procesor má chovat, jaký má instrukce, registry, ...

• HW architektura

- implementace nějaké ISA

• Instrukce - specifikuje je ISA

1. load instrukce z adresy v IP
2. decode inst.
3. load operands
4. execute inst.
5. store result
6. increment IP

• Třídy instrukcí

- load: mem. → reg.
- store: reg/imm. → mem. } slow
- move: reg → reg
- aritmetika + logika
- skoky - podmíněné × nepodmíněné
  - přímé = přímo adresa kam
  - nepřímé = adresa kam je na adrese
  - relativní = o kolik se posunout

x86 má HW zásobník ← • call + return

→ nebo se zásobník dělá přes registry

- při volání funkce je potřeba zásobník, kam si ta funkce uloží svou návratovou adresu
  → odkud byla zavolaná

- **Registry**
  - obecné – umí aritmetiku, adresaci a skoky
  - integer aritmetika
  - float aritmetika
  - adresové – pro nepřímou adresaci ve skoku
  - branch – pro skoky
  - příznakové = flags – booleovské hodnoty
  - predikátové – 1-bit registry pro každou instrukci – určují, zdali se provede
  - aplikační – speciálně určené pro nějakou sadu instrukcí
  - systémové – nastavení vlastností toho procesoru
  - vektorové – paralelně dělá 1 instrukci na více dat

if-else rozhodnutí skoky

- **Jména registrů**
  - 99% je pojmenovaných přímo (EAX, r01)
  - dřív občas relativně vůči vrcholu zásobníku

- **Aliasing registrů**
  - překrývání registrů → x86: EAX

  - snaha se tomu vyhýbat → složité překladače

- **x86**
  - → není ortogonální ⇒ specifické instrukce pracují se specifickými registry
    EAX = akumulátor, EBX = base, ECX = count, EDX = data
  - → ortogonální ISA = libovolná instrukce může použít libovolný reg (všeobecné)
    ↳ třeba očíslované r0–r31 a jsou ekvivalentní
  - → segmentové registry
  - → flags + IP, má HW stack

- **MIPS**
  - → r0–r31 = 32-bit obecné registry, ale některé mají special význam
    - r29 = stack pointer – ukazuje na vrchol zásobníku
    - r30 = frame pointer } slouží pro volání funkce
    - r31 = return address → link registr pro jal instrukci – místo caller
    - r0 = zero – je vždy nula
  - → nemá HW stack ani flags
  - → když se zavolá fce, tak instrukce jal skočí na tu fci a do r31 zapíše kam se vrátit
    současně někam na zásobník uloží aktuální stav registrů a slibuje, že
    je po návratu obnoví – kromě nějakých temp. registrů → preserve registry
    ↳ na return value je nějaký special registr

- Instrukce MIPS — x86

a ← b op c — a ← a op b

pevné kódování — proměnlivá délka instrukce

⇒ 1 instrukce = 32 b.

- ABI = Aplication Binary Interface
  - → specifikuje, jak a na co se mají které registry používat
  - → většinou to vydává autor ISA
  - → všechny compilery to musí dodržovat, jinak by nebyly kompatibilní
  - → říká aliasy registrů
  - → jaké registry jsou preserve = volání fce je nezmění

- Příznaky
  - nemají je všechny ISA
  - běžné jsou zero, sign a carry
  - jsou systémové a uživatelské příznaky — x86 je má promíchané — musí se to složitě maskovat
  - x86 zrada — ne všechny instrukce nastavují všechny příznaky,... → postupně se sem přidávaly
- Klasifikace instrukční sady → dnes už jen kvůli zpětné kompatibilitě
  - CISC — komplexní sada = spousta speciálních instrukcí ⇒ hodně tranzistorů
    - ↳ ty instrukce se pak překládají do mikrokódu z mikroinstrukcí
  - RISC — redukovaná sada = jenom ty základní instrukce → ale na 1 simple instrukci je 1 mikroinstrukce
    - ↳ rychle se to dekóduje + stačí málo tranzistorů
  - VLIW — Very Long Instruction Word ⇒ třeba 128 bitová instrukce
    - ↳ ta instrukce se nemusí dekódovat ⇒ fast — třeba v síťových switchích
  - EPIC — Explicitly Parallel Instruction Computer
    - ↳ v instrukcích je explicitně co má běžet paralelně — dnes to dělá HW

→ Load-Execute-Store = instrukce pracuje jen s registrama a nesahá do paměti (třeba aritmetika)

- Překlad instrukcí
  - instrukce dekóduje assembler → ale textovému zápisu instrukcí se taky říká assembler

- Co všechno je v procesoru?
  - řadič paměti → z něj vede sběrnice do paměti
  - hierarchie keší
  - jádro / jádra
    - registry
    - logické procesory = uvnitř 1 jádra je víc proudů instrukcí najednou → intel → hyperthreading

- Zjednodušené schéma
  - 1 jádro má:
    - výpočetní jednotku ~ execution unit
    - cache – 3 úrovně
      - ~ 10kB L1 → 4 takty → skoro stejně rychlá jako registry + dělení na instrukční / datovou
      - ~ 100kB L2 → 16 taktů → větší + o řád pomalejší + unifikovaný = kód i data
      - ~ 1MB L3 → 200 taktů → sdílená mezi jádry + zase o řád pomalejší
    - vlákna – víc vláken sdílí 1 výpočetní jednotku ∴ nebývá využitá na 100%
      - ale každé vlákno má svoje registry
  - dělá se prefetch = HW odhadování co se má načítat
  - L3 je ve skutečnosti rozdělovaná na L3 slices
    - → každé jádro má svůj a spojuje je obousměrný ring ← mega rychlá sběrnice
    - → přístup k L3 není brzy různě dlouho

- Schéma jednoho jádra → Intel Coffee Lake
  - Front End čte instrukce a dekóduje je
  - Coffee Lake má 5-cestný dekodér = dekóduje 5 instrukcí v 1 taktu
    - ↳ 4 jsou simple dekodéry na simple instrukce → přeloží se na 1 mikroinstrukci
    - ↳ 1 je komplexní na ty komplexní instrukce co se přeřádají do mikrokódu
  - po dekódování instrukce spadnou do „bazénu" a tam čekají na zpracování
    - ↳ Out of Order Execution – celkem chaotické "pool"
      - ↳ instrukce se probírají, ale pořád musí mít navenek stejný efekt
  - → v tom bazénu plavou ty mikroinstrukce, přičemž každá má nějakou „barvu" a čeká, až se jí otevře port příslušné barvy → proletí jím a vyřeší tu svou operaci
  - → každý port umí jiné instrukce
  - → nakonec to vlete do reorder buffer, kde se ty výsledky přeřádají do správného pořadí, aby ten výsledný efekt byl správný

• CPU pipeline
- vykonávání 1 instrukce se rozdělí na 14–19 fází = stages
- v 1 okamžiku mám rozdělaných více instrukcí a každá je v jiné fázi
  ↳ inst. fetch, inst. decode, execution, čtení z paměti, zapsání výsledku, ...
- každý takt se posunu o 1 fázi dál
- náběh pipeliny = když nabíhají první instrukce a ještě nepracují všechny fáze
- doběh reálně nenastane – to jenom v SW
→ když výpadne, tak musí znovu nabíhat – pomalé
⇒ podmíněné skoky se snaží CPU předvídat (jejich výsledek) } ≈ 95 % to odhadne správně
  ↳ při špatném odhadu musí celou pipeline zahodit ↳ napíše výsledky
→ návrh procesoru je jednodušší a overall to je rychlejší, ale vzniká latence
  ↳ musí přeběhnout celou novou fázemi

• Super-skalarita
- v každé fázi se nachází víc instrukcí současně
- 2-cestný procesor je 2× rychlejší než 1-cestný ← za jednotku času 2× víc instrukcí
- dnešní procesory jsou kolem 5-cestné
- můžou být asymetrické ⇒ víc simple a komplex instrukcí
→ když nastane konflikt (registrů, výsledků, ...), tak na sebe ty instrukce čekají

• Cesta instrukce

Pipeline decoder → pool → prolezu portem → reorder buffer

- **Paměťová hierarchie** → je na to potřeba upravit OS a programy
  - volatile → reg., cache, RAM
  - persistent RAM → non-volatile, rychlejší než SSD, CPU k ní má přímý přístup
  - externí paměť → SSD, flash disky, HDD → je potřeba řadič a sběrnice
  - archivní paměť → magnetické pásky

- **Adresový prostor**
  - → očíslovaný slovy ⇒ 32 bit adresový prostor znamená 2^{32} slov
  - → dnes 1 slovo = 1 byte
  - → fyzicky: byty ve 2D poli
    - → řadič paměti nejdřív vybere řádku a pak indexuje sloupci v ní
    - → pro jinou řádku ji musí odvybrat → to je pomalé
      ⇒ sekvenční přístup je nejrychlejší
  - → doba přístupu
    - CAS = # taktů než můžu adresovat další sloupec v dané řádce
    - RAS = čas na aktivaci řádky

- **Reprezentace dat**
  - unsigned int: $0 - 2^n - 1$
  - signed int: dvojkový doplněk: -2^{n-1} až $2^{n-1} - 1$
  - float: podle IEEE 754 → mantisa a exponent s biasem → CPU to pak převede na nějakou svoji reprezentaci
    ⇒ $V = (-1)^{sign} \cdot mantisa \cdot 2^{exponent - bias}$

- **Endianita**
  - → big endian = MSB first → ostatní + síťové protokoly
  - → little endian = LSB first → intel → LLL → LE má LSB na lowest adrese

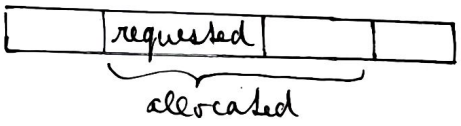
- **Data Alignment**
  - → datový typ velikosti X bytů musí ležet na adrese A: $A \% X = 0$
  - vnitřní zarovnání → každý typ ve structu je zarovnaný na násobek své velikosti
  - vnější zarovnání → structy v poli taky musí být aligned → je sám double? 20B ⇒ 24B
    ⇒ velikost toho structu musím zarovnat na násobek největšího typu v ní
  - → sizeof(struct) vrátí velikost včetně vnitřního i vnějšího paddingu
  - → tohle řeším jen pro základní typy ⇒ nebudu to řešit pro struct ve structu

- **Typy proměnných**
  - global - naalokují se při spuštění programu a smažou se při ukončení
  - local - alokované na zásobníku, vznikají v bloku { a končí v bloku }
  - dynamicky alokované - sám si je musím vytvořit → C++ malloc(), C# new
    ↳ instance tříd → sám je musím odebrat nebo to dělá garbage coll.

- **Alokace paměti**
  - → úkol je najít blok nevyužité paměti dostatečné velikosti
  - → spustím program → před tím, než se pustí main(), tak si runtime toho jazyka od OS vyžádá heap = velký souvislý úsek paměti
    → z toho heapu pak alokuju
  1) alokace - řeknu si o blok dané velikosti, dostanu adresu = pointer na tu proměnnou
  2) používání bloku
  3) explicitní uvolnění bloku / G.C.

  → heap si pamatuje zabranou paměť → po větších blocích - min. 64 B

- **Fragmentace paměti**
  - interní



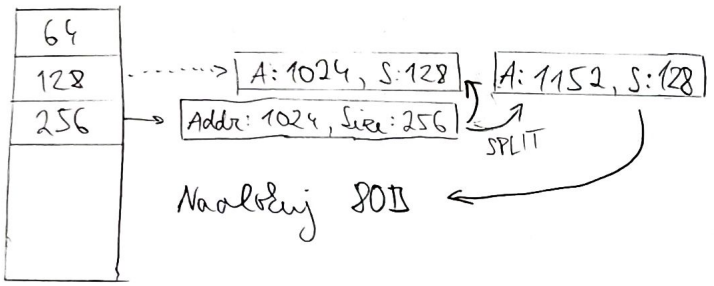
→ chci 48 B ⇒ dostanu 1 blok = 64 B

  - externí → mám hodně volného místa rozdrobeného do malých nesouvislých úseků

- **Dynamická alokace paměti**
  - → řeším alokuju věci s proměnlivou délkou
  - → jak si pamatuju volné bloky? ⌐ spoják volných bloků (adresa + velikost)
    ⌙ bitmapa bloků (1 bit 1 blok, málo rozlišuje velikosti)
  - → alokační algoritmy
    - first fit → vezmu první volné místo a rozštípnu ho
      ↳ easy & fast, ale může ničit velké bloky

    - next fit → pamatuju si, kde jsem skončil a z toho místa pokračuju
      ↳ trochu míň fragmentuje, ale musí si pamatovat, kde skončil
    - best fit → projde to celé a vezme ten nejmenší, kam se ještě fitne
      ↳ nechá velké bloky, ale vytvoří spoustu malých děr
      ⇒ na externí fragmentace může být ve výsledku ještě horší
    - worst fit → projde to celé a dá to do největšího bloku
      ⇒ očividně nejhorší, ničí velké bloky

- **Buddy memory allocation** – alokační alg. co se actually používá
  - → bloky o velikosti 2^n ⇒ adresy alligned na 2^n
  - → mám pole spojáků bloků o velikostech 64, 128, ... 16 kB
  - → když mám naalokovat 128 B blok a mám nějaký → prostě ho dám ✓
    - ↳ pokud ho nemám, tak najdu nejmenší 2^n blok kam se to vejde
  - ⇒ ten rozštěpím na 2 buddies, jeden si nechám a druhý naalokuju / štěpím dál
  - → těch spojáků je konstantně moho ⇒ alokace je konstantní



- **Dealokace** ⇒ pokusím se je mergnout
  - ↳ adresa těch buddies se liší jen o 1 bit – ta velikost → chci ho otočit
  - ⇒ adresa kamaráda je Addr XOR Size

→ dealokace není konstantní ∵ musím projít spoják pro daný Size
→ externí fragmentace je dobrá, ale má to velkou interní fragmentaci

- **Cache**
  - = datová struktura, kde mám data co často používám / dlouho se počítají
  - → HW ale i SW implementace
  - → pokud cache dojde, tak musím vybrat oběť ⇒ stránkovací algoritmy → pozdější
  - → když zažádám o data co sám nejsou, tak se fetchnou a příště to bude fast

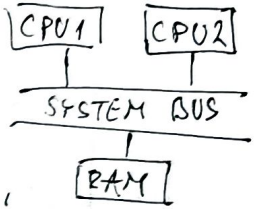
(Obecná 'cache')

- **Cache v procesoru**
  - → spoléhá na **lokalitu přístupu** = většinou přistupuju k datům co jsou vedle sebe
    - ⇒ předvídá si ty data ⇒ schovává latenci přístupu do RAM
  - → při requestu prohledává od nejrychlejší L1 po nejpomalejší L3
  - → **Koherence řeší** = problém když více jader přistupuje ke stejným datům
    - ⇒ ta data se musí přesunout do té sdílené L3 cache
  - → když nějaké jádro používá nějaká data hodně, tak jsou hot a jdou do L2 → L1

- **Terminologie řeší**
  - **cache line** = 1 blok v řeši (není organizovaná po B) – standardně 64 B
  - **cache hit** = když requestím data, co už jsou nakešovaný ~ 95 %
  - **cache miss** = když ty data ještě nemám nakešovaný ⇒ cache line load
  - **cache line load** = načítám data z paměti, pokud je cache plná ⇒ oběť
    - ↳ pokud jsem oběť v řeši zmodifikoval, tak ji musím aktualizovat v paměti
  - **cache line state** – používá se MESI protokol → 4 stavy řeš lajny

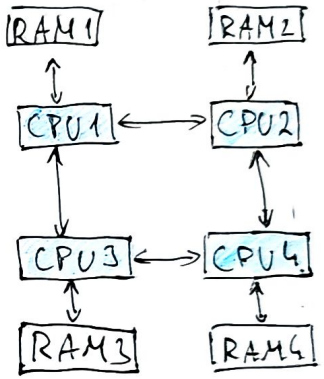
- **Asociativní paměť**
  - → RAM čipy ocitalizují cache linama ⇒ dvojice (klíč, hodnota)
  - → je to vlastně HW slovník ⇒ mega fast, ale stojí to HODNĚ tranzistorů
  - → snaha je, aby cache byla plně asociativní
    - ↳ většinou je třeba 4-asociativní = 4 různý cache linky mají stejný klíč
      - ↳ stejný bity té adresy, ale vznikají kolize

- **Systémy s více procesory**

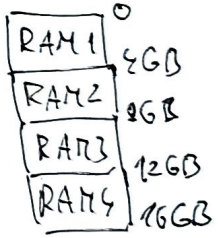


1) **SMP** = Symetric MultiProcessing
   - → 1 sběrnice mezi sdílenou RAM a CPUs
   - → simple, ale pro více procesorů je ta sběrnice úzké zablokovaná

2) **NUMA** = Non-Uniform Memory Access
   - → každý procesor má svůj kus RAM
   - → ty RAMky musí být dostupné i z ostatních CPUs
   - → jsou navzájem propojené → je to mega drahý
   - → CPU mají 1/2/3 piny pro ta rychlou sběrnici
   - → doba přístupů do těch RAMek je různá ⇒ **NUMA factor**
   - → ty RAMky tvoří 1 společný adresový prostor
   - → té dvojici RAM—CPU se říká **NUMA uzel**



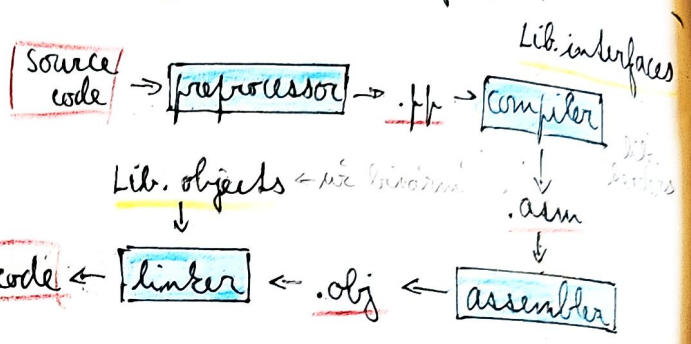
Address space



- **Programovací jazyky**
- **Překladač**

→ formálně to je zobrazení slov ze vstupního jazyka generovaného gramatikou do výstupního jazyka generovaného jinou gramatikou / přijímaného automatem (CPU)

→ jazyk má nějaká pravidla a lexikální elementy (while, do, for, ...)

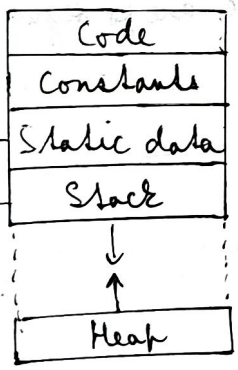


1. **Preprocesor**: zpracuje zdroják a příkazy pro něj → C: #include, #def, makra, ...
   ⇒ vytvoří .ii soubor – stále textový zápis
2. **Překladač**: dostane .ii a rozhraní knihoven ⇒ .asm assembler texták
3. **Assembler**: z .asm vyrobí binární .obj → to binární forma zápisu
4. **Linker**: až takhle zpracuju všechny zdrojáky na .obj, tak to spolu s objektama knihoven dostane linker a vyrobí výsledný executable

→ dneska se 1.2.3. dělá na 1 spuštění překladače

→ OS pak při spuštění udělá z programu proces
⇒ musí načíst do paměti nějakou DS s daty toho procesu

- **Organizace paměti při běhu programu**



→ read-only, smí se tam pouštět instrukce
→ read-only
→ globální proměnné, alokace při spuštění ⇒ statická adresa
→ lokální proměnné, rekurze, volání funkcí
← stack a heap rostou proti sobě
→ dynamická alokace proměnných, společná pro všechna vlákna

segment := kus paměti co má stejný vlastnosti → segment kódu, konstant, ...

- inicializovaná s.d. → int a=5; → je to inicializované už v tom executable souboru
- neinicializovaná s.d. → int a; → inicializace až při spuštění programu
  ↳ při spuštění, ještě před main, se pustí defaultní konstruktor / se to vynuluje

→ stacky pro vlákna → každé vlákno volá funkce ⇒ potřebuje vlastní zásobník
↳ stack má omezenou velikost (~10 MB) ⇒ není možná rekurze do nekonečna
⇒ není možné obrií lokální pole, ...

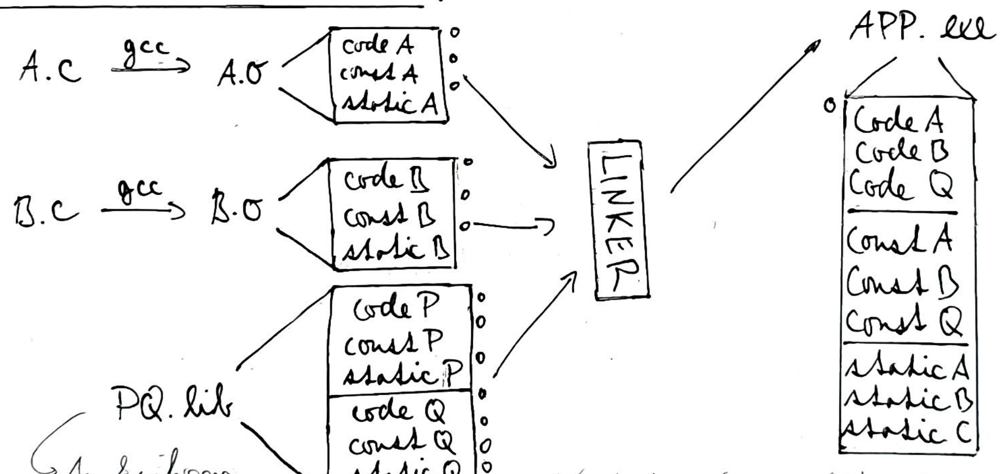
- **Knihovny**
  = kolekce zkompilovaných zdrojových modulů v 1 souboru → std. lib. C je 1 soubor, vlastně jen víc objektů
  - **statické** (.lib) – 1 soubor z více modulů a linker z ní přímo okopíruje ty potřebné části do toho výsledného executable souboru
  - **dynamické** (.dll) – do toho executable souboru si jenom poznamenám, že potřebuji nějakou fci z této knihovny a až za běhu to sežene loader, nahraje do paměti a upraví příslušné adresy v tom executable

→ statická vyrábí větší .exe, ale ty funkce tam jsou zabetonované → vydrží to [illegible]
→ pokud všechny programy využívají nějakou std. lib., tak dává smysl dynamická

- **Linkování**
  → linker posbírá všechny ty přeložené objekty + s. knihovny + rozhraní d. knihoven
  → vyrobí ten výsledný executable pro daný OS → ten z něj umí vyrobit proces

- **Loader**
  → část OS, co má za úkol načíst ten executable a dynamické knihovny do paměti a spustit to

- **Co všechno linker dělá?**

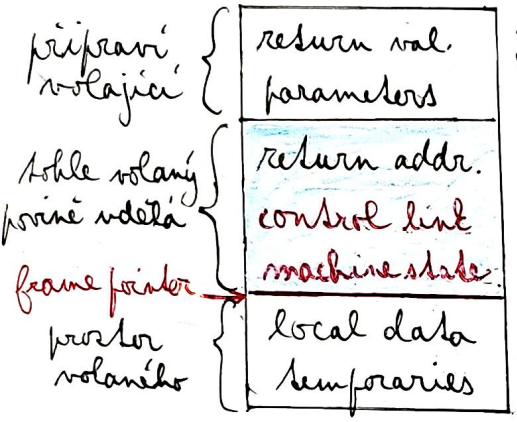


→ slepování není přímo za sebe, ale po segmentech
→ nekopíruje celou knihovnu
↳ ta knihovna obsahuje více objektů
→ dává tam jen použitý věci ⇒ import math

→ musí se taky řešit to, že před linkováním nevím, jakou budou mít nakonec funkce, konstanty, … adresu ⇒ překladač do všech těch instrukcí zapíše relativní adresu od začátku segmentu, ve kterém ta věc leží
  → překladač do toho objektu píše relokace – pamatuji si instrukci + ten segment
⇒ linker ty instrukce musí po slepení opravit → pro linker je bázová adresa toho souboru 0
  ⇒ ke každé té relativní adrese přičte bázovou adresu toho jejího segmentu po slepení
⇒ loader při spuštění dělá další relokace – vůči bázové adrese toho executable
→ linker hledá entry point = object, kde to začne běžet → není to main(), ale std. lib.
→ knihovna publikuje jaké má fce, konst., … ⇒ v .obj sekce public
→ zdroják říká, že chce externí fci ⇒ v .obj sekce extern
} → klíčové slovo public; taktéž se nakonec zavolá main
⇒ linker postupuje rekurzivně od entry pointu a pro externy hledá publiky ⇒ přidává moduly

- **Běhová podpora jazyka** = Run-time → to je potřeba pro překlad
  - statická – compiler + realizace rozhraní knihoven
  - dynamická – konstruktory a destruktory globálních objektů } to je potřeba pro běh
    - organizace paměti + obsah paměti před spuštěním
    - volací konvence funkcí
    - dynamické knihovny toho jazyka

- **Volání funkce** — stack frame
  - → funkce při zavolání dostane aktivační záznam a v něm pracuje
    - ↳ nikam jinam nesahá – kromě globálních proměnných



- → malé věci / reference se vrací registry
- ↳ pokud je to nějaký velký struct, tak volající připraví tohle místo → volaný dostane pointer
- → kam se pak má vrátit
- → frame pointer aktivačního záznamu volající funkce
- → stav registrů co se musí zachovat
- → lokální proměnné
- → když dojdou registry

je frame pointer reg. co obsahuje frame pointer aktivačního záznamu

- → tenhle celý aktivační záznam je na zásobníku

- **Volací konvence**
  - → řídí se jí všechny překladače, aby to bylo kompatibilní
  - **public name mangling** → linker jen hlavně porovnává řetězce publicu a externů
    - "mandlování jmen" ↳ znetvoření / přidělání
    - ⇒ aby fungovalo např. přetěžování / více konstruktorů
  - musí se domluvit kdo co uklízí z toho zásobníku + **call/return sekvence**
  - předávání parametrů → na stacku / registry + v jakém pořadí se na zásobník dávají ↳ co se volajícímu pro účely stane
  - návratová hodnota → buď registrama / přes stack ↳ je samozřejmě zpráva dobrá
  - registry – které se musí zachovat (**preserve**) a které ne (**scratch**)
    - ↳ jaké jsou role registrů

- **Call/return sekvence** = posloupnosti při volání
  - → volající předává parametry a po návratu volané funkce je i smaže
  - → volaná funkce smaže svoje lokální data, obnoví machine state, nastaví frame pointer reg. na control link, vrátí se a smaže tohle všechno } jako
  - → mazání = posunutí pointeru na vršek zásobníku
  - → existují volající konvence, kde funkce volaná uklízí i ty parametry → jde to jen pro přesný počet parametrů

- **Předávání parametrů** → čisté C umí jen hodnotou
  - hodnotou – prostě kopie a pak je to pro tu volanou fci jako lokální proměnná
  - referencí – předává se adresa, v C++ pomocí & → fce ví, že to není lokální var
    ↳ na ten stack se dá ta adresa → typ instance používají nepřímou adresaci
  - → čistý C řeší předávání referencí tak, že hodnotou předá pointer
    ↳ rozdíl: pointer není fixní + musím explicitně psát ta *, takže je to otrava

- **Proměnné**
  = pojmenovaný kousek paměti, který drží nějakou hodnotu
  → má nějaký typ (ve staticky typovaných jazycích)
  → kde leží?
  - C/C#:
    - static data – globální proměnné → alokace při spuštění
    - stack – lokální proměnné → alokace na zásobník při běhu programu
    - heap – dynamické proměnné → alokace na heap pomocí new (C#) / malloc (C)
  - python:
    - slovník – (klíč, hodnota) = (jméno proměnné, info o proměnné)
      ↳ tohle používá třeba python, javascript,... ↳ typ, hodnota, adresa,...

- **Heap**
  → dynamická paměť, kterou si při výpočtu můžu brát
  → není to ten stack, takže se to při návratu z funkce nemaže jako lokální proměnné
  → funkce volaná může něco dynamicky naalokovat a vrátit fci volající pointer
  - **alokace** – evidence volných bloků, alokační algoritmy
    → je to explicitní → např. new List<> → tohle vrátí pointer
  - **dealokace** – C, C++ jde explicitně udělat free / delete → vyžaduje mi to se o to fakt starat, a ten free se nezavolá ⇒ na heapu to pak zůstane
    ↳ zapomínalo se na to + musely se řešit exceptions
    ↳ **memory leak** = nemám GC a ztratím všechny pointry na tu paměť
  C#, Java
- **Garbage collector** = automatická dealokace, dodela pomalé algoritmy → mega fast alokace
  ⇒ GC se nezavolá dokud nedojde paměť ⇒ nemusím řešit alokační algoritmy
  → GC je pomalý + není možné alokovat když běží ⇒ freeze
  → hodně programů spoléhá na to, že se GC nikdy nezavolá → odstraňuje fragmentaci
  ⊕ → heap consolidation = po tom co GC doběhne, tak ty aktivní bloky „seskupí" na stranu...
  ⊖ → ovlivňuje to výkon, může dojít k zamrznutí v nepředvídatelný moment
- **GC strategie** – reálně se používají pozměňující
  - **trasování** (= tracing) → projde aktivační záznamy a najde živé proměnné → z nich to jde dál rekurzivně → mužou to být nějaké objekty
    → nakonec smaže všechno co není označené jako živé
  - **počítání referencí** → když udělám novou lokální referenci ⇒ ++count
    ↳ když ta lokální proměnná s ton referencí zanikne ⇒ --count
    → když count = 0, tak udělám free
    → pozor na cykly / odlehlé komponenty souvislosti – trasování tenhle problém nemá

- **Přenositelnost kódu**
  - → aby to fungovalo na více OS, překladačích, architekturách, ...
  1) **CPU architektura** → ale je možné si vynutit, že sekvence int bude 32 bit
     - C, C++ → různé velikosti typů, třeba int je podle architektury
     - C#, Java → fixed velikosti typů → int je všude 32 bit
     - → někdy je třeba řešit i endianitu dat
  2) **překladač** → gcc, msvc, clang → třeba deaktivovat barvení jenom syntaxe
     - → C++ má více překladačů s různými special featurami
     - ⇒ řešení: používat syntaxi a knihovny ze standardního jazyka
     - → překladače se snaží dodržovat ISO standard toho jazyka
     - → třeba C# nebo Java mají jen 1 překladač ⇒ ten problém tam není
  3) **OS** ≈ systémové volání
     - → různé sys-cally, ale funkce to má stejné → OPEN, ...
       - ↳ jen různá jména a interfacy (někdy to může vypadat stejně)
     - → podmíněný překlad = ve zdrojáku označím co se má přeložit podle OS
  - → Java a C# tohle všechno dělají sami za runtimu ⇒ je to samé od sebe přenositelné

- **Přenositelnost pomocí Virtual Machine**
  - → Java: bytecode, C#: CIL
  - → Java, C#, se přeloží do instrukcí procesoru co neexistuje ⇒ **mezikód**
  - → ten mezikód dostane nějaký nativní VM, co je za runtimu interpretuje (ten runtime už je součástí VM)
  - → ↑ bezpečnost a kontrola → zavře se to do sandboxu a ten interpret kontroluje, jestli to nedělá něco co nemá
  - → řešení problému rychlosti
    - **Just-in-Time** = JIT → jednotlivé funkce, třídy, ...
      - → postupně za běhu se mezikód překládá na nativní kód a to se ukládá na disk
      - ⇒ nemusím to překládat dvakrát - když volám poprvé nějakou fci, tak se přeloží podmínky než se zavolá přeložení
    - **Ahead-of-Time** = AOT
      - → přeloží se to celé při instalaci a spustí se to už přeložené

• Operační systémy
→ není přesná definice, ale má 2 hlavní úkoly

1) abstraktní stroj = abstrakce nad HW
→ reprezentovaný kernel API → systémové volání (jsou schované v nějaké knihovně) — to bývá napsané v C
→ schovávají před programátorem tu HW implementaci a poskytuje to C rozhraní

2) resource manager = přidělování prostředků procesům
→ managuje všechen HW a přiděluje aplikacím HW prostředky
→ ty aplikace to musí nějak sdílet mezi sebou
→ alokace (mem), time-sharing (CPU), abstrakce (disk, síť, grafika ...)

• Režimy procesoru — Když se aplikace pokusí udělat systémové [illegible]
1) uživatelský mód → pro všechny aplikace, nemá přístup k nějakým reg. + instrukcím
2) kernel mód → používá ho OS nebo jen nějaká jeho část - zbytek běží v user
↳ kernel = ta část OS co běží v kernel módu → [illegible]

• přechod kernel → user : OS to dělá triviálně - je na to special registr
• přechod user → kernel → např. některé sys-cally potřebují kernel režim OPEN PRINTF

entry pointy do kernel režimu = způsoby jak to přepnout
↳ jsou jasně definované a OS je kontroluje → třeba na to je special instrukce
↳ po přepnutí ta aplikace skočí na nějaké jasně definované místo v paměti, kde pro ní OS připravil ty instrukce
↳ parametry pro ten sys-call (jméno souboru) se předávají registry
→ entry point může být i to, že se ten program pokusí udělat nějakou systémovou instrukci, což CPU pozná že je chyba a předá řízení OS
↳ ta sys instrukce je nějak definovaná pro ten přechod

Architektura OS

- **monolitická** → takhle se dělaly první OS → Linux kernel
  - → velká složitá věc, která celá běží privilegovaně
  - → nemá to pořádnou strukturu – kromě entry-pointu, ze kterého je nějaká volba servisních procedur co pak vedou dál na utility procedury
  - ⊕ velmi efektivní, hodně rychlé sys-cally
  - ⊖ v tak velkém programu prostě budou chyby + když se někdo dostane do OS, tak může dělat co chce a má přístup k celému počítači
  - ⊖ původně se to ani nedalo rozšiřovat, dnes je možné načítat dynamické moduly ⇒ je tam ještě větší prostor pro chyby
- **vrstevnatá** → evoluce těch starých monolitů
  - → organizace OS do vrstev s různými službami a oprávněními
  - → vrstva n+1 může využívat pouze služby vrstvy n
  - ⊕ bezpečnější a jednodušeji rozšiřitelné
  - ⊖ návrh rozhraní mezi vrstvami je velmi náročný
- **mikrokernel**
  - → snaha je udělat kernel co nejmenší → třeba míň než 1 MB
  - → nad tím kernelem jsou jednotlivé moduly (filesystem, řadič disku, ...)
  - → kernel zajišťuje komunikaci mezi těmi moduly
    - → posílání zpráv jako mezi klientem a serverem
  - → když nějaký modul spadne, tak ho kernel zase spustí ⇒ nezabije celý OS
  - → kernel i ty moduly mají svojí protected paměť, do které si nevidí
  - ⊕ bezpečné, spolehlivé, snadno rozšiřitelné
  - ⊖ je to hodně pomalé – kvůli posílání těch zpráv
  - → windows se snaží být mikrokernel
  - → tím mikrokernel už ani nezávisí na architektuře CPU, protože ještě pod ním je Hardware Abstraction Layer

- **Zařízení** = HW kousek co slouží k nějakému účelu
  - **device controller** = řadič → řídí to zařízení
    - → elektricky komunikuje s tím zařízením, řeší signály na drátku, A/D konverzi ...
    - → těch zařízení k němu může být připojených víc ⇒ mají nějakou topologii

HW

  - **device driver** = ovladač → 1 driver bude pro nějaký druh zařízení, ne pro 1 konkrétní
    - → část OS, komunikace už probíhá pomocí instrukcí CPU → zapíše něco někam do paměti, aby OS vědělo, že se něco stalo
    - → poskytuje abstraktní rozhraní pro vyšší vrstvy OS

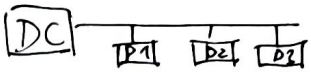
SW

- → při bootu (BIOS, UEFI) se projdou zařízení a každé dostane nějakou adresu
- → info o těch zařízeních se zapíše do tabulky, kterou pak dostane OS

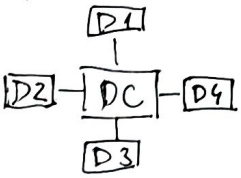
• Topologie zařízení

DC = Device Controller

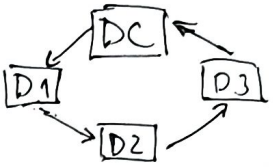
- **sdílená sběrnice** → hlavní driver
  - → musí se řešit arbitrace pro hodně zařízení
  - → přístup dat je omezený ⇒ pro hodně zařízení špatné



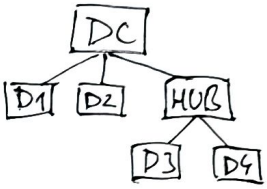
- **star** → dnes u disků
  - → zařízení je p2p připojené k řadiči
  - → nevýhoda je složitější konstrukce řadiče



- **ring** → moc se nepoužívá
  - → orientovaná kružnice se zařízení
  - → levnější řešení, velká latence



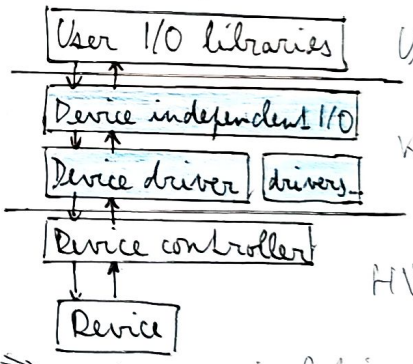
- **tree**
  - → DC je v kořeni, může se větvit pomocí hubů
  - → řadič i ovladač musí být schopni zvládat změny topologie toho stromu



• Komunikace se zařízením = Device Handling

- → sobě řeší OS, většina kódu OS je nezávislá na ovladačích a řadičích
- → programátor → v C může udělat přímo sys-call, ale pro přenositelnost je lepší použít std. lib.
- → pro otevření souboru zavolám funkci fopen(), která udělá syscall OPEN
  - ↳ OS dá tomu souboru **handle** = ID, podle kterého bude identifikace při READ
  - → ta read žádost se dostane až k ovladači
  - → ovladač dá request řadiči toho disku
  - → ta HW komunikace mezi řadičem a diskem může probíhat víckrát než se požadavek vyřídí
  - → až budou data ready, tak dá ovladači ke standardní knihovně adresu na ně → ta ji dá mě
- → problematická je ta část, kdy ovladač čeká na ty data

čekání



⇒ domluva mezi ovladačem a řadičem

- **polling**
  - → CPU periodicky kontroluje stav toho zařízení a čeká až bude hotovo
  - → musím aktivně čekat + to má latenci kvůli periodě
- **interrupt**
  - → zařízení pošle CPU signál, že už má hotovo ⇒ CPU přeruší aktuální proud instrukcí a skočí do **obsluhy přerušení** – ten to vyřídí a vrátí se
  - → řadič je drátem připojený k CPU pro ten signál ⇒ CPU na to potřebuje pin
- **DMA** = Direct Memory Access
  - → DMA řadič dokáže přesunout ty vypočtený data z řadiče toho zařízení do paměti
  - ⇒ až budou data v paměti, tak se vyvolá interrupt
  - → **scatter/gather** → DMA dokáže z nesouvislých dat vytvořit 1 blok nebo naopak ten blok rozsekat → třeba když přijde packet, tak ho rozsekám na TCP header, IP header a ty actuall data

HW skočí na nějaké pevné místo

dá zpět IP

CPU nemusí nic dopisovat ⇒ skočí se do obsluhy

⇒ nebo opačně do zařízení

- Typy přerušení = interrupts
  - externí → z venku přes nějaký pin na procesoru
    - ↳ je možné ho zamaskovat = zakázat / povolit → v kernel módu
  - exception = výjimka
    - → neočekávaně vyvolaná nějakou špatnou instrukcí → CPU ji vyvolá sám
    - → u všech těch výjimek je definované co se má stát když nastanou
      - ⇒ třeba se skočí na nějakou pevnou adresu
    - → využití pro emulaci nových instrukcích na starých procesorech
      - ↳ když přijde ta instrukce, tak nějam skočím, spočítám to a vrátím se zpět
    - fault = ta instrukce není schopna doběhnout
      - ⇒ udělá se rollback a to přerušení se udělá před tou instrukcí
    - trap = ta instrukce se provede a až pak nastane to přerušení
  - softwarové
    - → speciální instrukce co vyvolá přerušení ⇒ skočí do OS
    - → slouží jako entry-point OS pro sys-cally

→ pro hw přerušení je reálně nějakej řadič, který CPU posílá číslo toho přerušení pomocí nějakého protokolu ⇒ CPU pak skočí do obsluhy přerušení
→ adresy těch obsluh jsou buď fixní nebo v interrupt table

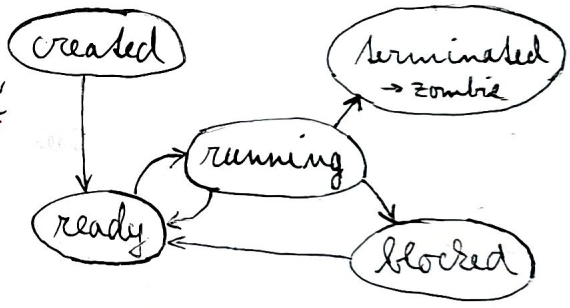
- Processing → OS řeší pouštění programů
  - program —(spustím / sys-call)→ proces → vlákno = stream instrukcí toho procesu
    - program: pasivní kód + data
    - proces: PID
    - vlákno: aktivita procesu
  - process = instance programu, je to nějaký objekt OS
    - → vytvoří ho OS pomocí sys-callu
    - → OS mu poskytuje prostředky a dělá si jejich evidenci, aby si je mohl vzít zpátky až ten proces skončí
  - vlákno = thread
    - → 1 proces může mít více vláken a běžet paralelně
    - → je to základní jednotka kernel schedulingu
    - } každé vlákno má vlastní zásobník
    - kontext procesoru = DS pro ukládání stavu vlákna
      - ↳ pamatuje si stavy registrů → je tam i stack pointer a další věci
    - → když CPU přepíná vlákna, tak musí udělat context-switch
  - fiber
    - → ještě lehčí jednotka plánování než vlákno
    - → taky má svůj kontext a zásobník
    - } OS o něm neví, dělá se to v user-space
    - → umožňuje kooperativní plánování
      - ↳ fiber 1 řekne „já už nechci běžet" a vybere další fiber

- **parent-child procesy**
  - → proces může spustit další proces a protože zná jeho PID, tak počkat až doběhne
  - → dítě si pamatuje PID rodiče → PPID → když skončí, tak mu pošle signál
  - → sirotčí procesy nechtěně jeho děti ⇒ adoptuje je init proces PID 1
    ⇒ zabít strom procesů je celkem těžké

- **Scheduler** = část OS co přiděluje výpočetní zdroje vláknům ⇒ plánovací algoritmy
- **Multitasking** – na 1 jádru běží více procesů současně ⇒ je tam nějaké přepínání
- **Multiprocessing** – mám více procesorů / jader ⇒ pro ten scheduler je to náročnější
  - → **affinita** procesu = snaha aby tohle vlákno běželo pořád na stejném jádru
    → tvůrci běží toho jádra
- **Real-time scheduling** → jiná verze schedulingu pro nestandardní situace
  - RT process má začátek (release time) a konec (deadline)
    ⇒ musí doběhnout někdy v tomhle intervalu → tohle je mega složitý naplánovat
    - hard deadline → nemá cenu to dodělávat, když jsem to nestihnul
    - soft deadline → pořád to má cenu dopočítat

  ⇒ třeba airbagy v autě nebo řízení jaderné elektrárny

- **Stavový diagram plánování**



  - → vlákno = unit of scheduling = **jednotka plánování**
  - → to vlákno má svoje stavy
    - **created** → čeká to na spuštění (release time)
    - **ready** → schopný běžet, ale nemá přístup k CPU
    - **running** → pokud už běží moc dlouho, tak zpět do ready
    - **blocked** → čeká na zdroje – třeba čtení ze souboru / user input zombie stav
    - **terminated** → už skončil, ale rodičovský proces si ještě nepřečetl výsledek
      - v tomto stavu si OS bere zpět všechny prostředky

- **Multitasking** → mám 1 jádro a chci na něm střídat vlákna → jak na to?
  - **kooperativní** → OS se o to nestará a všechny vlákna spolu spolupracují
    → možné jenom ve speciálních systémech
  - **preemptivní**
    - → každé vlákno dostane **time slice** (třeba 20 ms)
    - → OS má nějaký externí HW zdroj přerušení → timer
    - → pokud to vlákno během toho time slicu neskončí nebo se nezablokuje, tak na konci time slicu dostane interrupt a jde do stavu ready
    - → OS udělá context-switch a pustí tam jiné vlákno

- Scheduling – cíle
  - maximalizovat využití CPU a jader → pokud to potřebuju → většinou tolik tolik nechci
  - spravidlivě rozdělit výpočetní zdroje mezi vlákna
  - maximalizovat propustnost = # vláken co dokončím za jednotku času
  - minimalizovat dobu jednoho vlákna
  - minimalizovat čekací dobu v ready stavu
  - minimalizovat čas na odpověď v interaktivních aplikacích
    ↳ když kliknu na okno, tak musí rychle změnit stav blocked → ready → running a něco udělat
- Priorita procesu
  = číslo co stanovuje důležitost procesu → a vlákna tu prioritu většinou dědí
  → priorita je součet statické a dynamické priority
  - statická – přidělí se při spuštění procesu a může se pouze snižovat
  - dynamická – jednou za čas se ready vláknům inkrementuje
    ↳ když přejde do stavu running, tak se vynuluje
- Plánovací algoritmy
  - nepreemptivní ~ kooperační → strašně simple
    - First Come First Served – fronta, vypočítám vlákno a jde další
    - Shortest Job First – musíme předem vědět horní odhad execution time
      ↳ maximalizuje propustnost = throughput
    - Longest Job First
  - preemptivní
    - Round Robin → jako FCFS
      → je tam prioritní fronta + každé vlákno má time slice
    - Multilevel feedback queue → dynamický algoritmus
      → několik front podle toho jak dlouhý timeslice ty vlákna chtějí
      → horní fronta má nejmenší timeslice a spodní největší
      → pro výběr procházím fronty zeshora dolů dokud nenajdu neprázdnou frontu
      → pokud se vlákno zablokuje před koncem timeslicu tak ho dám výš
      → pokud na konci timeslicu pořád počítá, tak níž
      ⇒ interaktivní procesy skončí v nejvyšší vrstvě → většinu času jsou zablokované
  - Completely fair scheduler – CFS → Linux scheduler
    → stručně: vybírám proces co ještě neběžel moc dlouho a timeslice ~ doba čekání
    → ∀ proces si pamatuje svůj virtual runtime = jak celkově dlouho byl ve stavu running
      → taky se nějak bere v úvahu priorita toho procesu
    → ∀ jádro má červeno-černý strom s ready procesy → indexace podle VRuntime
    → vybírám nejlevější uzel = proces s nejmenším VRuntime
    → timeslice = čas čekání / # procesů ve stromě → to jsou ty ready vlákna

- Komunikace mezi procesy
  - → OS schovává prostředky procesu před ostatními procesy (např. paměti)
  - → kooperující procesy si chtějí nějak bezpečně posílat zprávy
    - pipes → něco jako síťové sokety ⇒ na jedné straně dávám data a na druhé je beru
    - sdílená paměť → dělá se to přes sys-cally
    - signály → používá je i kernel

- Soubory (z pohledu OS)
  - = data co spolu nějak souvisí / abstraktní proud dat
  - → kernel OS nerozumí sémantice těch dat

- Identifikace souboru
  - → filesystem používá číselné ID
  - → soubor má jméno a cestu k němu skrz nějakou stromovitou strukturu
    - → tohle tam je pro lidi + některé části jména mohou mít speciální význam (např. .git a .exe)

- Operace
  - open → najde soubor a načte si jeho metadata + vrátí file handle (0, 1, 2 = stdin, stdout, stderr)
    - ↳ handle je číslo, které kernel dokáže pomocí tabulky přeložit na ID toho souboru
  - close → uloží změny, odstraní ten objekt metadat a vyndá z tabulky file handle
  - read + write → používáme ten file handle, write se dělá do paměti a až pak na disk
  - seek → dělá se abstrakce ukazovátka, ale seek se jenom lineárně posouvá

  ⇒ každej proces má vlastní tabulku otevřených souborů – je v tom objektu proces

  - create, truncate, delete
  - flush = zapsání změn co jsou načtený v paměti na disk
    - → volá se náhodně a při zavření souboru
  - změna atributů

- Atributy souboru → jméno, velikost, typ, práva, timestamps, ve kterém sektoru běží (extension napovídá OS/aplikaci co to je; adresa na disku)

- Buffering → předčítání souborů, protože to je slow
  - OS: načítané sektory se nečtou dvakrát
  - C, C++: běhová podpora si to taky sama kešuje
  - sekvenční čtení → OS by sektory kešuje dopředu = read ahead

- Sync I/O: read() zablokuje vlákno, přečte, vrátí → vlákno ready
- Async I/O: read() zahájí operaci ale hned se z toho syscallu vrátí
  ⇒ vlákno běží dál a OS souběžně čte z toho souboru
  → pokud to nepřečte včas, tak se to vlákno dočasně zablokuje

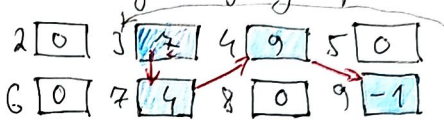
- **Adresář** = sada souborů
  - → většinou reprezentovaný jako soubor s nějakým special typem
  - → hlavně je to user-friendly a pomáhá to při hledání open()
  - → někdy si pamatuje nějaké atributy souborů v něm → podle filesystému
  - → je nějaká hierarchie s rootem
  - → operace: create, delete, rename file/subdir, search for name, list members
- **Uložiště souborů**
  - → na disku (externí paměť), RAM, v síti → musí tam být nějaký filesystem
  - → **virtuální soubory** → OS pomocí nich poskytuje nějaké funkce navíc /dev/null
- **File links**
  - **link (hard link)** – více položek v FS ukazuje na stejná data → stejné abs. file ID
    - → většina operací je transparentních, ale občas je to problematické
  - **symlink (soft link)** – soubor jehož obsah je cesta k jinému souboru
    - → je potřeba explicitně říct „follow symlinks"
- **File system** = datová struktura v uložišti a adresářů
  - → řeší jak a kde jsou data uložena + poskytuje abstrakci souborů pro OS
  - → musí umět:
    1) přeložit jména souborů na file ID
       - → rozkouskuje si cestu a rekurzivně se volá na podadresáře ⇒ slow
    2) pamatovat si ložiska dat souboru = sekvence bloků
    3) management volných bloků
       - → spoják, bitmapa,...

    blok = několik **souvislých** sektorů
    4kB = 8 · 512B

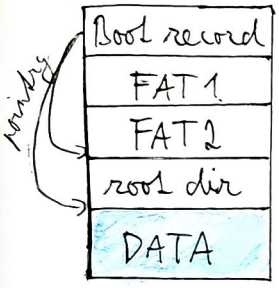
  - ⇒ prázdný textový soubor zabírá 1 blok = 4kB
  - lokální FS → na disku → FAT (win), NTFS (win), ext2/3/4 (linux), XFS (linux), BTRFS (linux)
  - síťový FS → protokol pro přístup k souborům přes síť → NFS (linux), Samba (win) } mohou být stejná rozšíření
  - → disk je možné rozdělit do **partitions** – každá tam může být různé FS
    - ⇒ na začátku disku je partition table → velikosti oddílu + FS tam
- **RAID** = Redundant Array of Inexpensive Disks
  - → způsob jak spojit více disků do jednoho → HW nebo to dělá OS
  - → pro velká data je potřeba (takovéhle) diskové pole
  - → je to rychlé a spolehlivé

    → HDD disků

- FAT = File Allocation Table
  - → hodně simple, z dob MS-DOSu
  - → je tam struktura FAT co se stará o volné bloky a pozici souborů
  - → na tom disku jsou 2 FATky (kopie) ⇒ záloha když počítač spadne při zápisu
  - → adresář je speciální soubor — na původním FATu 32 B — disaster recovery
    - → v něm je sekvence položek fixní velikosti s atributy
    - → pro každou položku si pamatuje číslo prvního bloku ⇒ ve FATce je spoják
  - → FATka je vlastně pole indexované čísly bloků a FAT[blok n] = blok n+1
    - ⇒ to pole začíná indexem 2 (0 a 1 mají nějaký speciál význam)
      - 0 = prázdný blok
      - -1 = poslední blok souboru



- → na disku je:

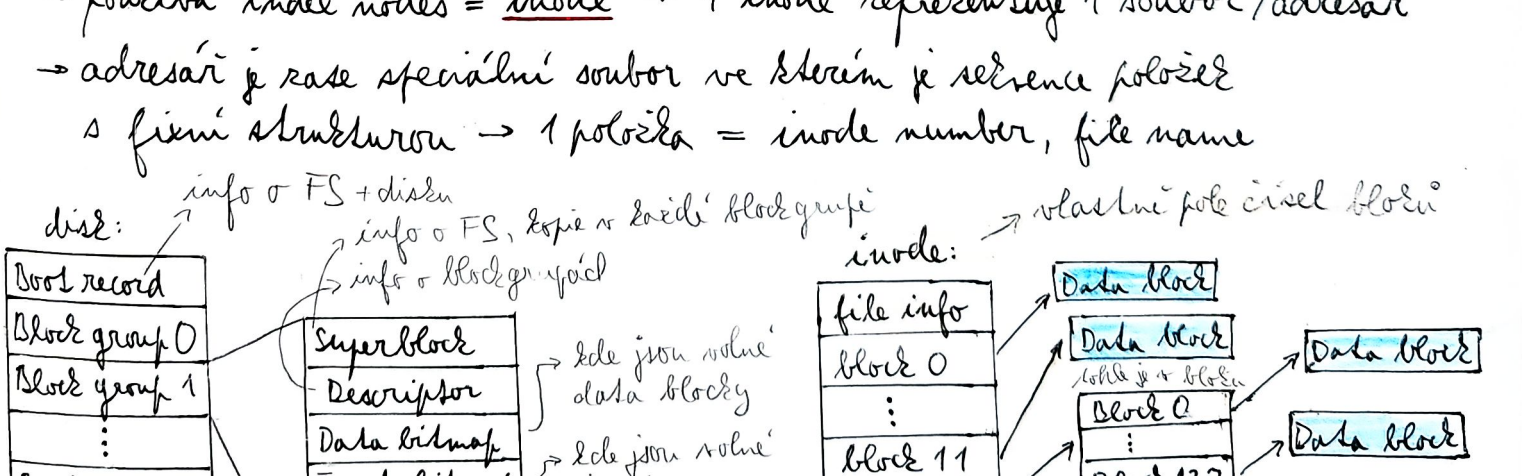


- → pro případ, že by se z toho disku bootovalo + info o FATkách a pointery na root a data
- → FATka má fixní velikost (je to jen pole)
  - ⇒ umím spočítat jak velká ta fatka bude a kolik bloků zabere
- → root má fixní velikost ⇒ omezený # položek
- ⇒ přístup k datům je simple protože všechno nad tím má fixní velikost

⊖ open je slow protože při prohledávání adresáře ho musím lineárně projít

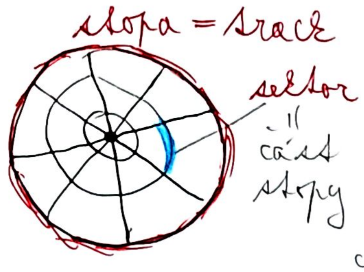
⊖ ten spoják je jednosměrný ⇒ neumím pořádně seek dozadu

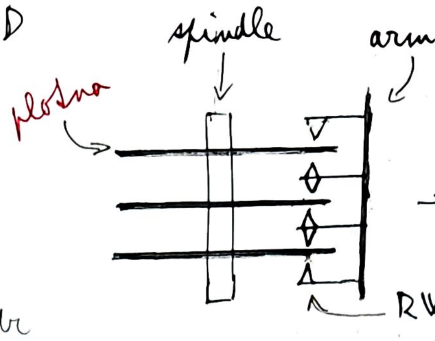
- ext2 – původně linux a boky celkem simple
  - → ext3 přidal journals aby se zlepšila disaster recovery
  - → ext4 umožňuje ukládat větší individuální soubory
  - → používá index nodes = inode → 1 inode reprezentuje 1 soubor/adresář
  - → adresář je zase speciální soubor ve kterém je sekvence položek s fixní strukturou → 1 položka = inode number, file name



- → náhodný přístup do souboru je OK
- → max. velikost souboru je $(12 + 128 + 128^2 + 128^3) \cdot 4\,kB \approx 8\,GB$

- Mechanika hard disků – HDD





1 sektor = 512B/2kB/4kB

→ uvnitř vakuum/helium

cylindr
=
cluster = stejná stopa na všech plotnách

blok = stejný sektor na všech plotnách

↳ tyhle bloky pak využívá FS

- flying height = vzdálenost hlavy od plotny ~ 5 nm
- rotační rychlost = 5400, 7200, 10k, 15k rpm (↳ rychlé; ↳ rychlejší to nejde; rozbitelné se to)

→ bottleneck je mechanický pohyb hlavy

→ složitá indexace sektorů ⇒ používá se LBA = Linear Block Addressing

- Disk scheduling

→ musíme rozhodnout pořadí vykonávání RW requestů

→ dřív to řešil OS, dnes to dělá disk

access time = seek time (→ pohyb hlavou ~ ms) + čas na rotaci + čas na transfer dat (→ zanedbatelné)

→ cíl je minimalizovat access time pro více I/O requestů

- FCFS = First Come First Served → ✓ pro malou zátěž (nepoužívá se)
- SSTF = Shortest Seek Time First → starvation = vzdálené requesty se nesplní
- SCAN = algoritmus výtahu → ✓ pro těžkou zátěž
  - CSCAN = Circular scan → dojede úplně nahoru a pak se vrátí do přízemí a cestou dolů nenabírá pasažéry → repeat
- LOOK / CLOOK – jako SCAN/CSCAN ale nenavštěvují konce disku ~ jako na Drofici
- FSCAN – 2 fronty, algoritmus pracuje s 1. a nové requesty dává do 2.

- Solid State Disk – SSD

→ bez pohyblivých částí, rychle se opotřebovávají → hlavně zápis a mazání (→ ty SSDčka)

→ organizace do bloků rozdělených na stránky → 1 page ~ 2-16kB (1 blok ~ MB)
↳ 1 block ~ 128-256 pages

→ read / write je po stránkách

→ erase je po blocích → ty stránky se označí jako invalid

⇒ když se rozbije 1 stránka, tak přijdu o celý blok

→ dělá se GC a data consolidation = odstraní se invalid bloky & validní stránky se setřesou na stranu

→ jsou speciál FS co to tak rychle neničí

• **Virtuální paměť** → používají je i instrukce i kernel OS

→ procesy pracují s virtuálními adresami do virtuálního a.p. = VAS

→ operační paměť má fyzický a.p. = PAS, fyzická adresa = 1 číslo

→ překládá se to hardwarově (⇒ nedělá to OS) v CPU pomocí MMU = Memory Management Unit

↳ je to zobrazení VAS → PAD, ale to mapování nemusí existovat

↳ pokud neexistuje, tak to MMU pozná a vyvolá výjimku typu fault

→ hlavní mechanismy – segmentace (outdated) a stránkování

→ procesy pracují s adresami (neví, že V) a při každém přístupu do RAM → převod HW

→ proč to děláme? (→ dnes už to není moc potřeba)

- hodí se větší adresový prostor (VAS může být větší než PAS)
  → přebytečnou virtuální paměť může OS odložit na disk
- bezpečnost – oddělím adresové prostory jednotlivých procesů
  ⇒ procesy si nemůžou sahat do paměti
  – je možné logicky oddělit segmenty VAS → read only, executable...

• **Segmentace**

→ koncepty:

- VAP je rozdělený na logické segmenty – stejně jako paměť běžícího procesu
  → segmenty jsou nějak očíslované – ≠ proces má vlastní číslování (čísluje se od 1)
- virtuální adresa má 2 části → číslo segmentu, offset v rámci segmentu
  → ty instrukce s tím nemají kernel pracovat
- operační systém spravuje **segmentační tabulku** – pro ≠ proces
  → je to pole indexované segmenty obsahující deskriptory těch segmentů
  → v deskriptoru je bázová fyzická adresa toho segmentu + délka + atributy (→ read-only, executable, systémová paměť)
  → OS jenom musí zajistit aby když běží tenhle proces byl někde v registru pointer na tu segmentační tabulku ⇒ CPU ví kde leží

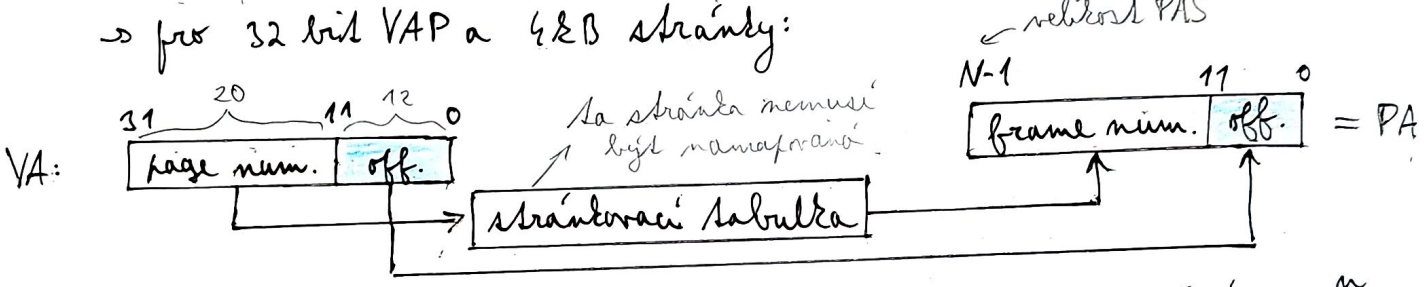
HW:
- ⇒ PA = ST[Segment num.] + offset → pokud offset ≥ délka segmentu ⇒
- → pokud je číslo segmentu out of range ⇒ vyvolá se výjimka segment fault
- → taky se kontroluje jestli třeba nedělám write do read only segmentu...

→ co když dojde prostor ve fyzické paměti?
  → nějaký segment se celý přesune na disk → v deskriptoru je present bit
  ⇒ pokud přistupuj do segmentu co není present ⇒ segment fault
  → OS ten segment načte z disku a jde se znovu

výpadek segmentu

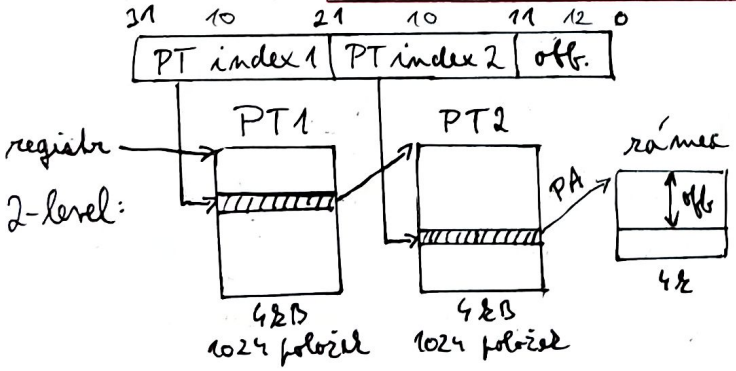
→ spolu se špatným managementem fyzické paměti (← vnější fragmentace) je tohle hlavní důvod proč už se segmentace nepoužívá

- **Stránkování**
  - → VAS je rozdělený na stejně velké části (stránky), velikosti 2^n
    - ↳ velikost VAS je rady 2^N → pro 32/64-bit CPU $N = 32/64$ – většinou
  - → PAS je rozdělený na stejně velké rámce = frame
    - → rámce mají stejnou velikost jako stránky → většinou $4kB = 2^{12}B$
  - → virtuální adresa je 1 číslo → narozdíl od segmentace
  - **stránkovací tabulka** – pole v paměti pro ∀ proces
    - → indexovaná číslem stránky → obsahuje číslo rámce + atributy
    - → v těch atributech je zase příznak P = present, co říká jestli mapování existuje (P)
    - → pokud mapování neexistuje / jiný problém ⇒ page fault = výpadek stránky
    - → pro 32 bit VAP a 4kB stránky:



    - → tohle funguje protože stránky a rámce jsou stejně velké a 2^n
    - → VAS a PAS můžou být různě velké
  - → 1 logický segment v tom procesu je rozdělený do více souvislých stránek ale ty rámce nemusí být ve fyzické paměti souvisle za sebou
    - ⇒ když dojde paměť tak nemusím vyhazovat celý segment ale jen 1 stránku

- **Problémy stránkování**
  - 1, **velikost** – pro 1-úrovňovou tabulku ↑ to je to pole
    - ↳ pro 32b VAS je 2^{20} 4kB stránek ⇒ $2^{20} = 1M$ položek
    - → pokud 1 položka = 4B, tak si musím pro ∀ proces pamatovat 4MB tabulku
    - ⇒ řešení: **víceúrovňová stránkovací tabulka** → ∵ nepotřebuju celý VAP



    - → v PT1 jsou fyzické adresy PT2 (page table)
    - → v PT2 jsou už fyzické adresy rámců
    - → ta tabulka má 4kB = 1 stránka
    - → PT1 musím mít vždy v paměti
    - → ty PT2 mapuju podle potřeby
  - → záznamy v PT1 mají pořád present bit ⇒ může být page fault na PT2
  - → 1 tabulka 2. úrovně stačí na nadresování $1024 \cdot 4kB = 4MB$ paměti

2, rychlost

→ překlad na PA vyžaduje víc přístupů do paměti ⇒ to je mega slow

⇒ nařešíme to do TLB = Translation Lookaside Buffer

→ TLB využívá asociativní paměť ⇒ slovník (stránka, rámec) (už bylo v keši)

→ využíváme toho, že procesy často přistupují ke stejným stránkám

• Algoritmus pro převod adresy u stránkování → HW

1. rozděl VA na číslo stránky a offset
2. chečni jestli to už není v TLB → pokud to tam je tak končím
3. projdi stránkovacíma tabulkama
   - rozděl číslo stránky na tolik částí kolik je úrovní
   - jdi do PT1 → pokud tam je adresa PT2 jdu dál (pomocí present bitu)
     → pokud tam adresa další tabulky není ⇒ page fault
   - z poslední úrovně vyzvedni číslo rámce a ulož ho do TLB
4. aktualizuj příznaky A = Accessed a D = Dirty ve stránkovací tabulce a TLB
   → A ~ stránku někdo použil, D ~ do stránky někdo zapsal
   → musím je změnit i když jsem se k té stránce dostal přes TLB
5. získej fyzickou adresu z čísla rámce a offsetu

• Jak se řeší výpadek stránky?

→ OS interrupt handler musí zjistit kde se stala chyba v jaké stránkování

→ ví který proces běží ⇒ v objektu procesu je pointer na tu stránkovací tabulku

→ možné důvody:
- neoprávněný zápis, čtení ze systémové paměti, ... (proces zkouší vzít segmenty a OS mu alokuje tu virtuální paměť)
- pokus o přístup do ještě nenaalokované virtuální paměti
- vše OK ale neexistuje mapování

→ jak vyrobit mapování?

→ OS najde volný rámec nebo vybere oběť algoritmem na výměnu stránek

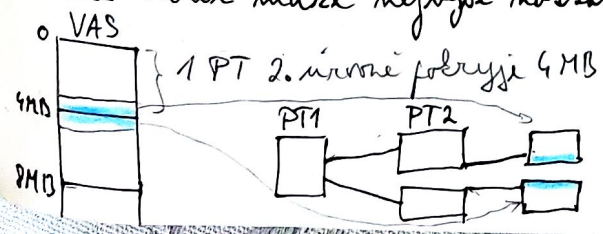
→ pokud je oběť dirty (změnila se) tak ten rámec uložím na disk

→ musím zrušit mapování oběti v TLB (→ TLB nemůžu změnit, to nabídá HW)

→ na volný rámec nahraju svůj obsah a opravím stránkovací tabulku

→ vrátím se zpátky z handleru a zkusím tu instrukci znovu

• Úloha: kolik může nejvíce nastat výpadků stránek když chci načíst 2 B?



→ 2 B můžou být na rozhraní 2 s.t. 2. úrovně

→ může nastat 4 výpadky!

→ je to nejenom na hraně stránek ale i tabulek!

- **Algoritmy na výměnu stránek**
  - → využívají se jakékoli situaci kde je potřeba vybrat oběť aby uvolnila místo
    - ⇒ vybírání rámce, cache, TLB

- **Optimal page algorithm**
  - → vybere stránku co bude nejdéle nepoužitá ⇒ nejmenší page-fault rate
  - → jen teoretický, je snaha se tomu přiblížit

- **Hodiny (clock)**
  - → rámce si zorganizuju do kruhu a dám tam ručičku – ukazuje na rámce
  - → využívá Access bit, který MMU nastaví na 1 když na tu stránku sáhne
  - → rámec vybírám tak, že postupuju ručičkou dokola:
    1. if A rámce ≠ 0: A ← 0 a jdu dál ← rámce nulluju
    2. if A rámce = 0: vyberu tenhle rámec ← dlouho nebyl použitý

- **NRU = Not Recently Used**
  - → příznaky A se nulují pravidelně – třeba 1 za minutu
  - → rámce rozdělí do 4 tříd používání podle příznaků A a D

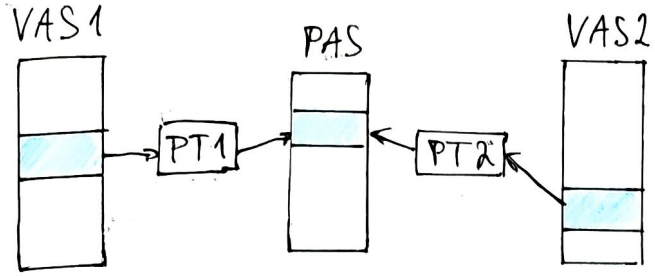
A	D	třída	
0	0	0	→ nepoužité + nezmíněné
0	1	1	→ nepoužité ale musím to zapsat na disk
1	0	2	→ někdo to používá
1	1	3	

  - ⇒ vyberu náhodný rámec z nejnižší neprázdné třídy

- **LRU = Least Recently Used** → tohle se reálně používá
  - → používám minulost pro předvídání budoucnosti
    - ⇒ vybere stránku na kterou nejdelší dobu nikdo nesáhl
  - → existují HW implementace
  - → **SW implementace:** → spoják nebo cokoliv
    - zásobník – když na něco sáhnu tak to dám nahoru a oběť je na dně
      - ↳ tohle je moc pomalé ⇒ aproximace NFU

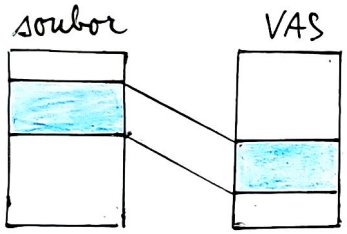
- **NFU = Not Frequently Used** – aproximace LRU
  - → každý rámec má počítadlo → někde bokem, do s. t. by se nevešlo
  - → pravidelně nuluju A a pokud A = 1 tak počítadlo inkrementuju
    - ⇒ vybere se rámec s nejmenším počítadlem → aby se hned nevyhodily
  - → novým stránkám musím na začátku dát nějaké skóre
  - → staré framy by se nevybraly ⇒ **aging**: periodicky count ← count/2

- **Sdílená paměť**
  - → více procesů spolu sdílí část virtuální paměti – způsob komunikace
  - → někdo udělá sys-call „vyrob sdílenou paměť" a další procesy se připojí



  - → každý proces má svoji s. t.
    a ta sdílená paměť je v VAP jinde
    ale ty rámce jsou sdílené
  - ⇒ ve sdílené paměti není možné používat ukazatele → jenom offsety
  - → při 2-level stránkování lze mít společnou tabulku 2. úrovně

- **Memory-mapped files** = paměťově namapované soubory
  - → normálně udělám open a pak dělám read a write
    - ↳ ten soubor se mi nakešuje někam do paměti ale pořád musím dělat R/W



  - → soubor si namapuju do virtuální paměti
    - ⇒ do toho souboru můžu přistupovat operacema procesoru load a store
    - ⇒ normálně nastavuju A a D bity
  - → když se ta stránka vybere jako oběť, tak pokud D ≠ 0, tak uložím
  - → problémy:
    - apendování na konec souboru → jsou tam nenamapovaný stránky
    - soubor může být větší než VAS

- **Virtual machine**
  - → přetížený výraz, teď se budeme bavit o virtualizaci ne Java/C# VM
  - → v rámci OS umožníme vyrobit chráněné prostředí,
    které umožňuje spustit další OS
    - ↳ je tam iluze toho, že to běží na reálném HW
  - → v případě problému se o to musí postarat ten **host OS** – ale většinou se to nedělje
  - → normální instrukce běží na CPU, ale speciál kernel instrukce (nastavování stránkování) emuluje ten host OS

⊕ **izolace** VM od zbytku systému
⊕ **enkapsulace** → VM ~ soubory ⇒ můžu dělat snapshoty kdyby to spadlo
⊕ **kompatibilita** → snadno se přenáší na jiný HW – VM poskytuje virtuální HW
↳ virtuální HW → virtuálky můžou mít nastavenou různou RAM, počet jader

- Kontejnery – virtualizace na úrovni OS
  - → zjednodušená virtualizace → používá se reálný HW ne virtuální
  - → OS kernel musí umožňovat existenci více oddělených user spaců
    - ↳ na linuxu si v kontejneru nepustím windows

Paralelní programování a synchronizace

- Paralelní počítání = používá se více jader a instrukce se provádějí současně
- Concurrent počítání = multitasking na 1 jádru
- Problémy paralelního počítání
  - Race condition
    - → když více vláken současně mění stejná data
    - ⇒ výsledek se může lišit podle časování / schedulingu OS
    - ⇒ není to deterministické
    - → příklad se spojákem:

```
LL l;
1. l.add(A)  →  A.next = head;
2. l.add(B)     head = A;
```

← interrupt

před assign dostane interrupt

a = 10

1. a = a+1
2. a = a+2

⇒ a ∈ {11, 12, 13}

  - Kritická sekce
    - → identifikujeme kritickou sekci kódu kde může vzniknout race condition
    - ⇒ řešení = mutual exclusion ⇒ v k.s. 1 vlákno at a time
  - Synchronizace – jak udělat mutual exclusion? = zachování integrity dat
    - → buď se kritická sekce zamyká nebo se vlákna nějak řídí
    - → realizace pomocí synchronizačních primitiv
      - aktivní – žerou čas procesoru (aktivní čekání)
      - pasivní (blokující) – kernel to vlákno zablokuje dokud není přístup povolen
        - ↳ pasivní se nemusí vždy vyplatit ∵ sys-call na blokování je drahý (10–100 instr.)
        - ⇒ pokud je race condition vzácná, tak je aktivní lepší
    - → aktivní potřebují HW podporu
      - → používají atomické instrukce → Test-and-set (TSL), Compare-and-swap (CAS)
        - ↳ instrukce co CPU zaručeně udělá celou a bez přerušení

int TSL: nastaví zámek na novou hodnotu a vrátí starou hodnotu

bool CAS: porovná zámek s danou hodnotou a pokud jsou stejné, tak ho nastaví na novou hodnotu, vrací true/false

- **Spin-lock** – aktivní
  - → když zámek = 0 tak je volno
    - ⇒ všichni se ho pomocí CAS snaží nastavit na 1
    - ⇒ až to vlákno dokončí kritickou sekci tak zámek nastaví na 0
  - → vhodné pro krátké čekání

- **Semafor** – blokující
  - → počítadlo a fronta čekajících vláken
  - → atomické operace UP a DOWN ale je to sys-call ⇒ provádí se to třeba pomocí spin-locku
  - → počítadlo se na začátek nastaví na # vláken co chceme najednou pustit do kritické sekce
  - → když vlákno chce do kritické sekce, tak zavolá DOWN
  - → když z ní odchází tak zavolá UP

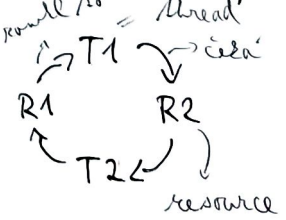
```
down:                                  up:
  if count > 0 : --count;                ++count;
  else:                                  if !queue.empty():
    queue.push(tohle vlákno)               v ← queue.pop()
    vlákno.block()                         v.unblock()
                                           --count;
```

- **Mutex** – implementace mutual exclusion
  - → atomické operace LOCK a UNLOCK
  - ⇒ binární semafor, kde na začátku count = 1

- **Bariéra**
  - → když vlákno dorazí k bariéře, tak se zablokuje
  - ⇒ čeká se až u bariéry bude určitý počet vláken a pak se všechny odblokují
  - → je to dobré na synchronizaci vláken – vím, že všechny jsou tady

- **Monitor** – v programovacích jazycích
  - → metody objektu nějaké třídy mají zámek jako Mutex
  - ⇒ na začátku metody je lock a na konci unlock
  - ⇒ 2 vlákna nesahají současně na stejný objekt

- **Deadlock** – vlákno 1 čeká až vlákno 2 dokončí operaci a to čeká až 1 dokončí operaci



  Coffmanovy podmínky:
  1. **mutual exclusion**: prostředky nelze sdílet
  2. **hold and wait**: vlákno drží prostředek & chce další
  3. **no preemtion**: prostředek nemůžeme vláknu jen tak sebrat
  4. **circular wait**: v modelovacím grafu je cyklus

- # Klasické synchronizační problémy

1, Producer - consumer

→ sklad o omezené kapacitě, výrobci zboží a konzumenti

- co když přijede výrobce a konzument zároveň? → musí se udělat mutex
- co když se sklad zaplní? → blokne výrobce – čeká na konzumenta → deadlock
- co když je sklad prázdný? → blokne konzumenta – čeká na výrobce

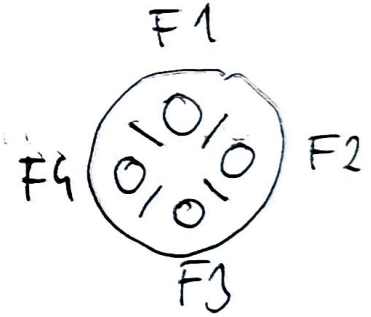
2, večeřící filozofové

→ N filozofů v kruhu – mají po pravé ruce vidličku a před sebou talíř

→ filozofové buď přemýšlejí nebo jí

⇒ jak se udělat aby každý někdy jedl

↳ co když dostanou hlad ve stejný moment?



- popadnu pravou a čekej na levou ⇒ deadlock
- když vlevo není, tak vrátím tu pravou ⇒ starvation

↳ pořád běží, nejsou zablokovaní, ale nenají se