

## C a C++

false = 0

true ≠ 0

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

→ implicitní konverze výsledků druhého typu ⇒ int · long = long

- array    int a[4] ← random hodnoty    → adresa pole = básová 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 n = 8;

int\* pw = &n;    & mi dala adresu

\*pw = 4;    \*pointer = přístup na danou adresu, mym n = 4

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

int n = 8;

int &pw = n;

pw = 4;    → n = 4

pw je mym jen jiné jméno pro n

- struct data {

char c;    1B

double d;    8B.

int i, j, k;    4B

}

- minimální a výšší rozdílnost alignment  
→ Typ velikosti X musí být na adrese A: A%X = 0

0B	c	↓	d	f
8B		↓	d	
16B	i		j	
24B	k	↓		

- výšší rozdílnost mezi velikostí největšího typu  
n si struktury  
aby se to doba dala do poří

void funkce(data in, data \*out) {

out->c = in.c;

out->d = in.d;

}

→ n c je něco předávané hodnotou

→ přidání referencí pomocí pointeru

→ přístup k memberům reference    | C++ reference  
(\*ptr).field = ptr->field    | data & out  
out.c = in.c

## Konstanty

const int jeden = 1; ← memória proměnná

constexpr int druhý = 2; ← compile-time konstanta - není v paměti ⇒ lepší

## preprocessing - zohle se řeší na úplném začátku kompilace

#include <stdio.h> → partii se tam nadpisuje obsah tohoto souboru

#define N 100

#ifdef ... } mgh

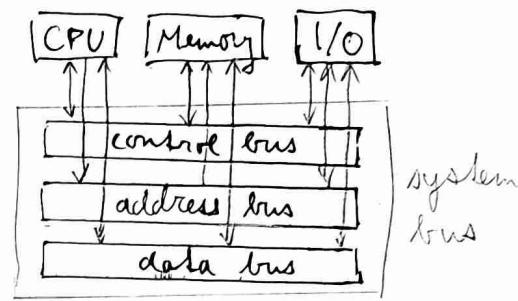
#endif

## CPU architektura

### Von Neumann

- program a data ve stejné operací paměti
- 1 sdílená sběrnice  $\Rightarrow$  vždy jen 1 transakce
  - $\hookrightarrow$  když CPU psá do paměti, tak nemůžeme  $\hookrightarrow$  paměti psát na monitor

formalý



### Harvard

- oddělena data a code memory  $\rightarrow$  sběrnice z CPU i do code mem
- počátek v různých řUC  $\rightarrow$  je potřeba více adresových prostorů pro procesory

$\Rightarrow$  trochu rychlejší

### Reálná

- CPU má paměťové řadiče a paměti jsou připojené vlastní sběrnici přímo k CPU
  - $\hookrightarrow$  dokonce má několik záhlaví do 1 paměti  $\Rightarrow$  paralelizace
- grafika je buď uvnitř procesoru nebo externí grafická karta
- South Bridge
  - připojený sběrnici k CPU
  - jsou na něj připojeny všechny periferie
    - $\Rightarrow$  HDD, DVD drive, myš, síťová karta, zvuková karta
- drív byl i North Bridge pro paměť  $\rightarrow$  formálně  $\Rightarrow$  jednalo dělá procesor
- všechny sběrnice jsou peer-to-peer = mezi dveřmi rozdělení (drív sdílená)

### Co je to architektura?

- instrukční sada - ISA - specifikace jak se ten procesor má chovat, jaký má instrukce, registry, ...  
 $\hookrightarrow$  x86, MIPS, ARM

- 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

### Tridy instrukcí

- load: mem.  $\rightarrow$  reg.
- store: reg / imm.  $\rightarrow$  mem.
- move: reg  $\rightarrow$  reg
- aritmetika + logika

### skoky - podmíněné $\times$ nepodmíněné

- první = první adresa kam

- nepřímé = adresa kam je na adrese

- relativní = o kolik se posunout

### call + return

- při volání funkce je potřeba zasobník, kam si ta funkce uloží svou návratovou adresu  
 $\hookrightarrow$  odkud byla zavolána

x86 má HW rozložník

$\rightarrow$  nebo se rozložník dělá

přes registry

## • Registry

- obecné - umí aritmetiku, adresaci a sloky
- integer aritmetika
- float aritmetika
- adresové - pro neprázdnou adresaci ve sčítce
- branch - pro sloky
- příznakové = flags - booleovské hodnoty
- prediktovací - 1-bit registry pro každou instrukci - určuje, jestli se provede
- aplikativní - speciálně určené pro nějakou sadu instrukcí
- systémové - nastavení vlastností toho procesoru
- vektorové - paralelně dělá 1 instrukci na více dat

if - else množství slok

## • Jména registrů

- 99% je pojmenovaných přímo (EAX, r01)
- druh obvyk relativaře má vícenásobnou

## • Aliasing registrů

- pokrývají registrů → x86: EAX 
- snaha se tvaru vyhýbat → složitější překladače

Ax

## • x86

- nemí ortogonální ⇒ specifické instrukce pracují se specifickými registry  
EAX = akumulátor, EBX = base, ECX = count, EDX = data

- ortogonální ISA = libovolná instrukce může použít libovolný reg (nemí)
- třeba očíslování r0 - r31 a jsou ekvivalentní

## • Segmentové registry

- flags + IP, má HW stack

## • MIPS

- r0 - r31 = 32-bit obecné registry, ale některé mají speciální význam

- r29 = stack pointer - ukazuje na vršek sásobníku
- r30 = frame pointer } slouží pro volání funkcí
- r31 = return address } link register pro jal instrukci ~ miejsce caller
- r0 = zero - je vždy nula

- nemá HW stack ani flags

- když se zavolá fce, tak instrukce jal sločí na tím fci a dr r31 kopíruje tam se vrátí současné někam na sásobníku uloží aktuální stav registru a uloží, že je po návratu obnoví - tedy nějakých temp. registrov
- ↳ na return value je nejaly special register

preserve register

## • Instrukce MIPS

$a \leftarrow b \text{ op } c$

je nejednoznačná

$\Rightarrow 1 \text{ instrukce} = 32 \text{ b.}$

$\times 86$

$a \leftarrow a \text{ op } b$

je jednoznačná délka instrukce

## • ABI = Application Binary Interface

→ specifikuje, jak a na co se mají být registry používávat

→ většinou je uvedeno autor ISA

→ všechny komputery ho musí dodržovat, jinak by nebyly kompatibilní

→ různé aliasy registrů

→ jaké registry jsou preserve = volání fung je nezmění

## • Příkazy

- nemají je všechny ISA

- binární jsou zero, sign a carry

- jsou systémové a uživatelské příkazy -  $\times 86$  je má promítané → musí se to postupně se tím přidávat

-  $\times 86$  sada - ne všechny instrukce zacházejí všechny příkazy, ... složitějsí masivní

## • Klasifikace instrukční sady

→ dnes už je kvůli rychlé kompatibilitě

• CISC - komplexní sada = sestava speciálních instrukcí ⇒ hodné tranzistorů

↳ ty instrukce se pak překládají do mikročísel & mikroinstrukcí

• RISC - redukována sada = jenom ty základní instrukce

↳ rychle se to dekóduje + stačí málo tranzistorů

→ ale na 1 simple instrukci je 1 mikroinstrukce

• VLIW - Very Long Instruction Word ⇒ některá 128 bitová instrukce

↳ ta instrukce se nemusí dekódovat ⇒ fast - některá v sítích switchů

• EPIC - Explicitly Parallel Instruction Computer

↳ v instrukcích je explicitně co má být paralelně - dnes ještě HW

↳ některá architektura

→ Load-Execute-Store = instrukce pracuje jen s registrami a nezáleží do paměti

## • Překlad instrukcí

- instrukce dekóduje assembler → ale některým zápisu instrukcí se říká assembly

## • Co všechno je v procesoru?

- řadič paměti → k něj vede sběrnice do paměti
- hierarchie řešení
- jádro / jádra
  - registry → intel → hyperthreading
  - logické procesory = uvnitř 1 jádra je více proudů instrukcí na jednom

## • Z jednoduššího schéma

- každé jádro má:

- výpočetní jednotku ~ execution unit

- cache - 3 úrovně

- ~ 1024 L1 <sup>4 kB</sup> → shora stejně rychlá jako registry + dílem na instrukčním datovém
  - ~ 1024 L2 <sup>16 kB</sup> → rychlá + o rád pomalejsí + unifikovaný = rád v data
  - ~ 1 MB L3 <sup>200 kB</sup> → sdílená mezi jádra + různé o rád pomalejsí
- vláčka - více vláček sdílí 1 výpočetní jednotku :: nebyvá využívána na 100%
    - ale každé vláčko má své registry

- dělá se prefetch = HW odhadování co se má načítat

- L3 je ve skutečnosti rozlosována na L3 slices

→ každé jádro má svůj a spojuje je obousměrný ring mega rychlá sběrnice

→ přístup k L3 řeší kvůli různé dlonky

## • Schéma jednoho jádra → Intel Coffee Lake

- Front End čte instrukce a dekóduje je

- Coffee Lake má 5-cestný dekódér = dekóduje 5 instrukcí v 1 kladu

↳ 4 jsou simple dekódery na simple instrukce → přebírá se na 1 mikroinstrukci  
↳ 1 je komplexní na ty kompletní instrukce co se překládají do mikrokódů

- po dekódování instrukce spadnou do „baceňů“ a tam čekají na zpracování

## • Out of Order Execution - celkově chaotické "pool"

↳ instrukce se provádzí, ale pořadí musí mít návratné slevující efekt

- v tom baceňu plavou ty mikroinstrukce, přičemž každá má nájdenou „barvu“ a čela, až se jí stane pořadí finálně barvy → pak již jde a význam tu svou operaci
- každý pořadí umí jiné instrukce
- nákoncě ho vše do reorder bufferu, kde se ty výsledky překládají do správného pořadí, aby ten výsledný efekt byl správný

## • CPU pipeline

- vykonávání 1 instrukce se rozdělí na 14-19 fází = stages
- v 1 okamžiku může být v rámci rozdělených několika instrukcí a každá je v jiné fázi
  - ↳ inst. fetch, inst. decode, execution, čtení z paměti, zapsání výsledku, ...
- každý krok se posunem o 1 fazu dál
- náhled pipeline = když nabíhají první instrukce a ještě nepracují všechny fáze
- dobeh reálně nenastane - to jenom v SW
- když vypadne, tak musí znova nabíhat - pomalu
- ⇒ podmíněně slohy se snaží CPU předvídat (jejich výsledek)
  - ↳ při řízeném odhadu musí celou pipeline zahrát
  - ↳ odhadné upřímné
  - ↳ např. cykly
- návrat procesoru je jednodušší a overall to je rychlejší, ale vzniká latence
  - ↳ musí přebít téměř všechna fáze

## • Super-skladatka

- každá fáze se nachází více instrukcí současně
  - ↳ za jednotku času
- 2-cestný procesor je 2x rychlejší než 1-cestný
  - ↳ 2x více instrukcí
- dnešní procesory jsou již 5-cestné
- můžou být asymetrické → viz simple a complex instrukce
- když nastane konflikt (registru, výkonu, ...), tak na sebe ty instrukce cekají

## • Cesta instrukce

Pipeline decoder → pool → prokres portem → reorder buffer

## Paměťová hierarchie

- volatile  $\rightarrow$  reg., cache, RAM
- persistent RAM  $\rightarrow$  non-volatile, rychlejší než SSD, CPU & mě má přímý přístup
- externí paměť  $\rightarrow$  SSD, flash disky, HDD  $\rightarrow$  je potřeba řádce a sběrnice
- archivní paměť  $\rightarrow$  magnetické pásky

jí ma do poříčka upravit OS a programy

## Adresový prostor

$\rightarrow$  očíslovaný slouy  $\Rightarrow$  32 bit adresový prostor znamená  $2^{32}$  slov

$\rightarrow$  dnes 1 slovo = 1 byte

$\rightarrow$  fyzicky: byly ve 2D poli

$\rightarrow$  řádky faměli nejdřív vybere řádku a poté indexuje sloupcem v ní

$\rightarrow$  pro jinou řádku ji musí odvysílat  $\rightarrow$  to je pomalejší

$\Rightarrow$  sekvencí přístup je nejrychlejší

$\rightarrow$  doba přístupu

$\bullet$  CAS = # kroků než musí adresovat další sloupec v daném řádku

$\bullet$  RAS = čas na aktivaci řádky

## Reprezentace dat

$\bullet$  unsigned int:  $0 - 2^m - 1$

$\bullet$  signed int: dvoukoty doplněk:  $-2^{m-1}$  až  $2^{m-1} - 1$

$\bullet$  float: podle IEEE 754  $\rightarrow$  mantisa a exponent s biasem  
 $\Rightarrow V = (-1)^{\text{sign}} \cdot \text{mantisa} \cdot 2^{\text{exponent} - \text{bias}}$

CPU když  
přesle na nějto  
zroji odpovídající

## Endianita

$\rightarrow$  big endian = MSB first

$\rightarrow$  little endian = LSB first

$\rightarrow$  LLL  $\rightarrow$  LE má LSB na Lowest adrese

## Data Alignment

$\rightarrow$  datový typ velikosti  $X$  bytů musí ležet na adrese A:  $A \% X = 0$

$\bullet$  vnitřní zarovnání  $\rightarrow$  každý typ ve struktu je zarovnán na násobek své velikosti

$\bullet$  vnější zarovnání  $\rightarrow$  struktury v poli taky musí být aligned

$\Rightarrow$  velikost celé struktury musí být zarovnána na násobek největšího typu v ní

$\rightarrow$  sizeof(struc) vrátí velikost vnitřního vnitřního i vnějšího padolu

$\rightarrow$  tedy řeším jen pro základní typy  $\Rightarrow$  nebudu ho řešit pro struktury struktury

## Typy paměti

- global - načítá se při spuštění programu a smírá se při ukončení
- local - alokováno na rámcovém, vznikají v bloku { a končí v bloku }
- dynamicky alokování - sám si je musíte vybrat → (+ malloc()), (# new instance břid) → sám je musíte odstranit nebo ho dělat garbage coll.

## Alocae paměti

- vždy je možné blok nevyužíti paměti dostatečné velikosti
- spuštění programu → před tím, než se spustí main(), tak si runtime kódu jazyka od OS vyzádá heap = velký souvislý úsek paměti → k tomu heapu pak aločujte → main() → D.C.

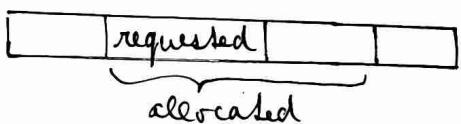
1, alocae - řeknu si o blok dané velikosti, dostanu adresu = pointer na tu proměnnou  
2, počítaný blok

3, explicitní uvolnění bloku / G.C.

→ nemusíte již zadávat

- heap si pamatuje zabranou paměť → pro některých blocích - min. 64 B

## Fragmentace paměti

- interní  → chci 48 B → dostanu 1 blok = 64 B

- externí → máme hodně volného místa rozdrobeného do malých nesouvislých částí

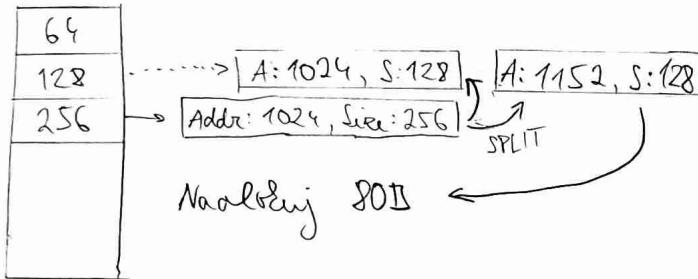
## Dynamická alocae paměti

- přišel aločujte vše s proměnlivou délou
- jak si pamatuju volné bloky? → spojí volných bloků → bitmapa bloků
- aločací algoritmy

adresa + velikost  
} blok blok nebo  
zobrazit → řešit

- first fit → neznej první volné místo a rozšíří ho → easy & fast, ale může mít velké bloky
- next fit → pamatuju si, kde jsem sloučil a k tomu místa pokračuju → trochu méně fragmentuje, ale musí si pamatovat, kde sloučil
- best fit → projde to celé a neznej ten nejménší, kam se ještě fitne → nechá velké bloky, ale vytvoří spousta malých dílů ⇒ ta externí fragmentace může být ve výsledku ještě horší
- worst fit → projde to celé a dá to do největšího bloku ⇒ očividně nejhorsí, mici velké bloky

- Buddy memory allocation - aloacií alg. CO se actually používá
  - bloky o velikosti  $2^n$  ⇒ adresy aligned na  $2^n$
  - main pole spojení bloků o velikostech 64, 128, ... 168
  - Edýr má méně možností 128 B blok a máme nejedný → prostředek ho dělit
    - ↳ pokud ho nemáme, tak majeme nejmenší  $2^n$  blok kam se to rozdele
  - ⇒ Jen rozdělíme na 2 buddies, jeden si necháme a druhý nabízíme / řešení dle
  - Tich spojení je konstantní mnoho ⇒ aloace je konstantní



- Deallocace ⇒ fokusujeme se je merou
  - ↳ adresa tich buddies se liší jen o 1 bit - svá velikosti → chci ho sloučit
- ⇒ adresa kamaráda je Addr XOR size

- deallokace není konstantní ∵ musíme projít spojení pro daný size
- externí fragmentace je dobrá, ale má to vliv na interní fragmentaci

## • Cache

- {
- = datová struktura, kde máme data co často používáme / dlouho se použijí
  - HW až i SW implementace
  - pokud cache dílce, tak musíme vybrat oběť ⇒ stránkovací algoritmy
  - Edýr začádám o data co sam nejsou, tak se feshuji a příště to bude fast

## • Cache v procesoru

- spočívá na lokalitu přístupu = některou přistupují k datům co jsou vedle sebe
- ⇒ přednosti si ty data ⇒ schovává lokenci přístupu do RAM
- při requestu prohledává od nejrychlejší L1 po nejomalajší L3
- coherence kesi = problém když více jáder přistupuje ke stejným datům
  - ⇒ ta data se musí přesunout do té společné L3 cache
- Edýr nejale jádro používá nejdelší data buňku, tak jsou hot a jdou do L2 → L1

## • Terminologie kesi

- cache line = 1 blok v kesi (není organizována po B) - standardně 64 B
- cache hit = Edýr requestem data, co má jsou nalezeny ~ 95%
- cache miss = Edýr ty data jisti nemá nalezeny → cache line load
- cache line load = načítání data z paměti, pokud je cache plná ⇒ oběť
  - ↳ pokud jsem oběť v kesi změnil, tak ji musím aktualizovat v paměti
- cache line state - používá se MESI protokol → 4 stavů kesi bajny

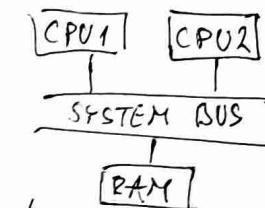
## • Asociativní paměť

- RAM když očísluje cache linama  $\Rightarrow$  dvojice (klíč, hodnota)
- je to vlastně HW slovník  $\Rightarrow$  mega fast, ale stojí to HODNĚ tranzistorů
- snaha je, aby cache byla plně asociativní
  - $\hookrightarrow$  reálnou je třeba 4-asociativní = 4 různý cache liny mají stejný klíč
  - $\hookrightarrow$  třetím bity té adresy, ale vznikají kolize

## • Systémy s více procesory

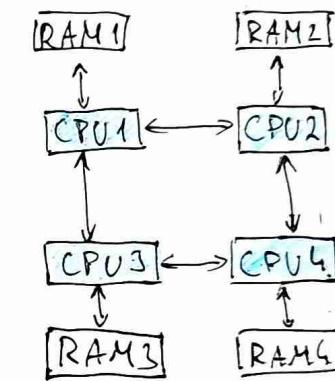
### 1) SMP = Symmetric Multi Processing

- 1 sběrnice má řízenou RAM a CPUs
- simple, ale pro více procesorů je ta sběrnice frotá zablokována

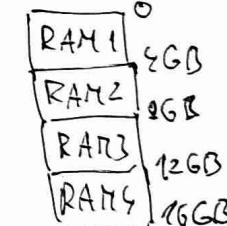


### 2 NUMA = Non-Uniform Memory Access

- každý procesor má svůj bus RAM
- ty RAMy musí být dostupné i z ostatních CPUs
- jsou nazývají propojené  $\rightarrow$  je to mega dráhy
- CPU mají 1/2/3 piny pro snadnou sběrnici
- doba přístupu do těch RAMů je různá  $\Rightarrow$  NUMA factor
- ty RAMy sdílí 1 společný adresový prostor
- když dvojici [RAM] — [CPU] se říká NUMA node



Address space



## • Programovací jazyky

### • Překladač

- formálně to je zobrazení slov ve vstupního jazyka generovaného gramatikou do výstupního jazyka generovaného jinou gramatikou / přijímaného automatem
- jazyk má nějaká pravidla a lexikální elementy (while, do, for, ...)

1. Preprocessor: spracuje sdroják a příkazy pro něj → C: #include, #def, macro, ...  
⇒ výsledný .pp soubor - stále faktory zápis

2. Překladač: dostane .pp a rozhraní environ ⇒ .asm assembly text

3. Assembler: z .asm vytvoří binární .obj → za binární formu zápis

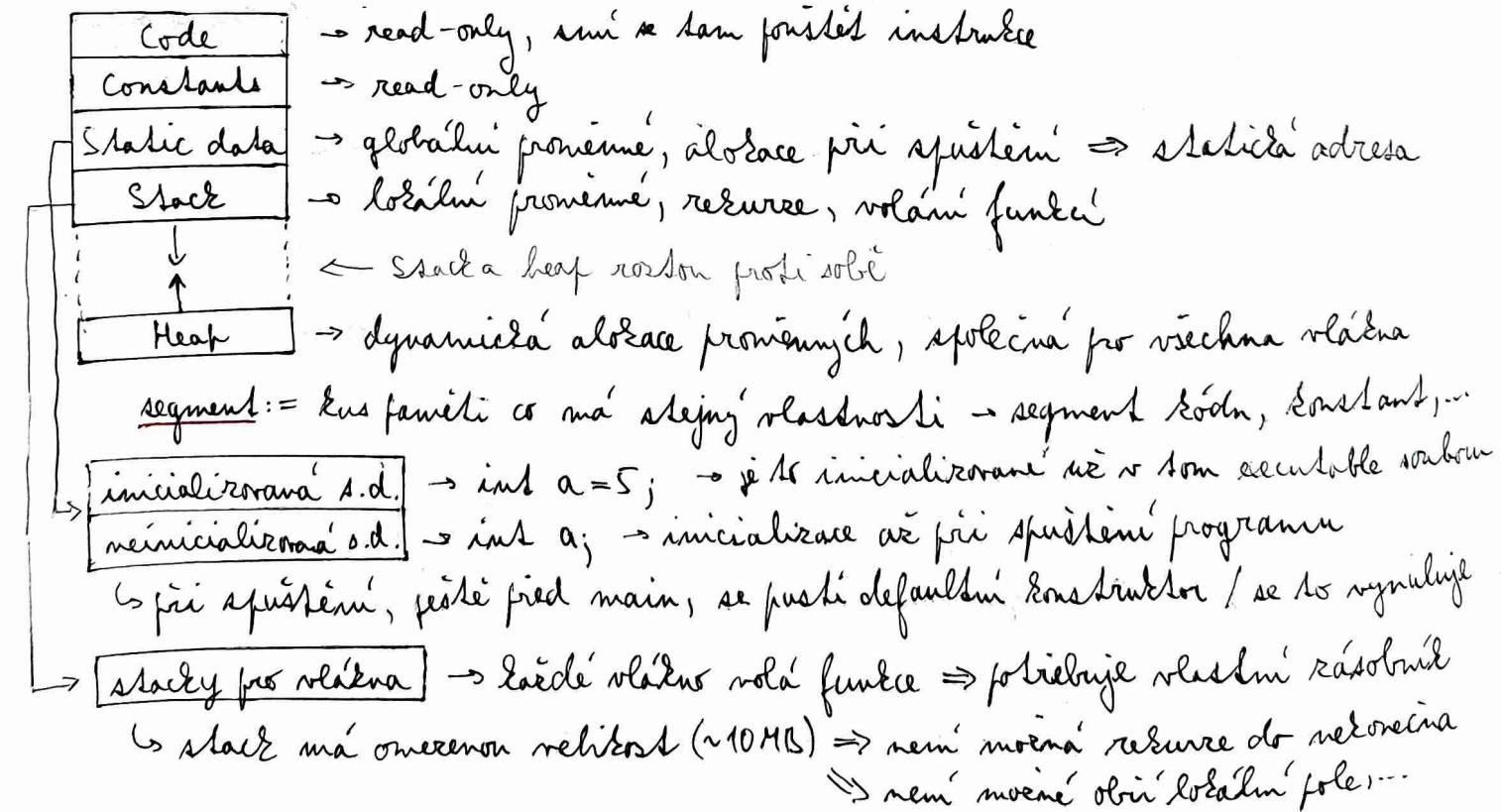
4. Linker: až tady spracuj všechny sdrojáky na .obj, takže spolu s objectama environ dostane linker a vytvoří výsledný executable

→ dneska se 1.2.3. dělá na 1 spuštění překladače

→ OS pak při spuštění udělá s programu proces

⇒ musí moci do paměti nějakon DS s dalým toho procesu

### • Organizace paměti při běhu programu



## • Knihovny

- = soubory s komplikovanými zdrojovými moduly v 1 souboru → std. lib. C je 1 soubor
- statické - 1 soubor s více moduly a linker z něj přímo odporuje . lib. ty potřebné části do toho výsledného executable souboru
- dynamické - do toho executable souboru si jenom poznamená, že potřebuji . dll. nějakou fci z této knihovny a až za běhu to se máne loader, nahraje do paměti a upraví příslušné adresy v tom executable

→ statická vyrábí něčí .exe, ale ty funkce tam jsou zabelonovány → výkon je neefektivní

→ pokud všechny programy užívají nějakou std. lib., tak dává smysl dynamika

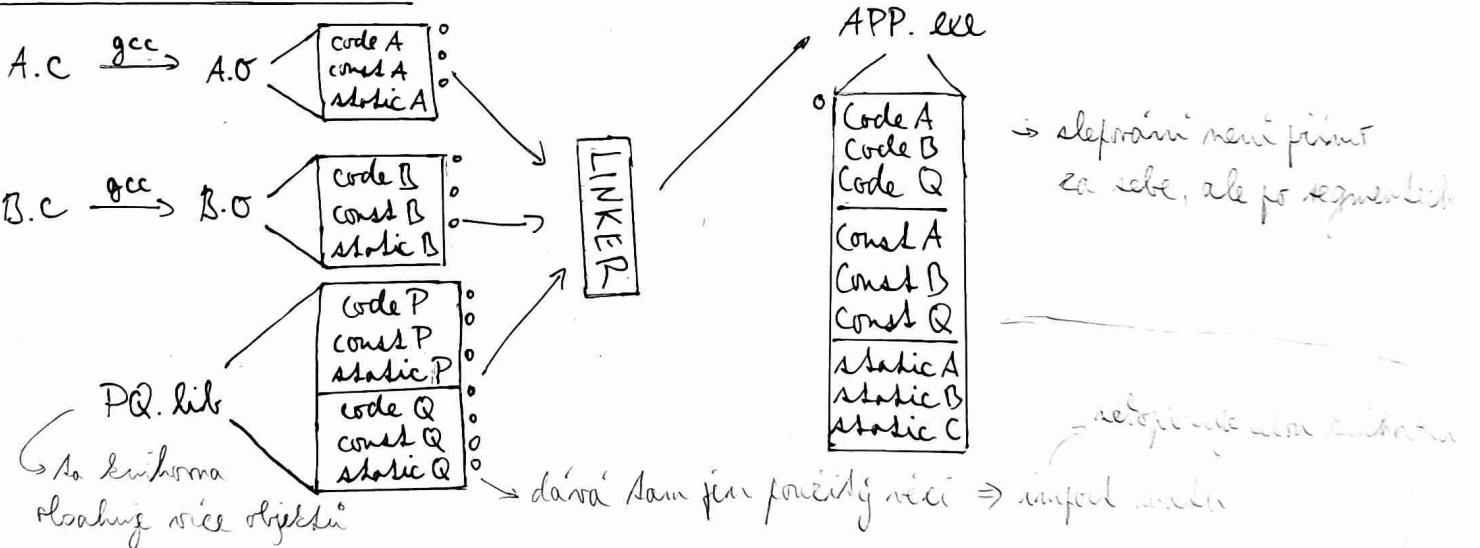
## • Linkování

- linker posbírá všechny ty přeložené objekty + s. knihovny + rozhraní d. knihoven
- vytvoří ten výsledný executable pro daný OS → ten je nejmenší rozložitelný

## • Loader

- část OS, co má za úkol načíst ten executable a dynamické knihovny do paměti a spustit ho

## • Co všechno linker dela?



- musí se toho řešit to, že před linkováním nemám, jakou bude mít například funkce, konstanty, ... adresu → překladač do nich těch instrukcí zapise relativní adresu od začátku segmentu, ve kterém ta věc leží
- překladač do toho objektu píše relocace - formátuje si instrukci + ten segment
- ⇒ linker ty instrukce musí po slopení opravit → před linkem je bárová adresa dole v tabulce
- ⇒ ke každé té relativní adrese příčle bárovou adresu toho jejího segmentu po slopení
- ⇒ loader při spuštění dela další relocace - něčí bárové adresy toho executable
- linker hledá entry point = objekt, sde to začne běžet → nemá to main(), ale std. lib.
- knihovna publikuje jisté funkce, konstanty, ... ⇒ v. obj. se má public { → všechny slouží public
- zdroják říká, že chce externí fci ⇒ v. obj. se má extern { všechny se mají vlastní jména
- linker postupuje rekurezivně od entry pointu a pro externy hledá publiky ⇒ přidává moduly

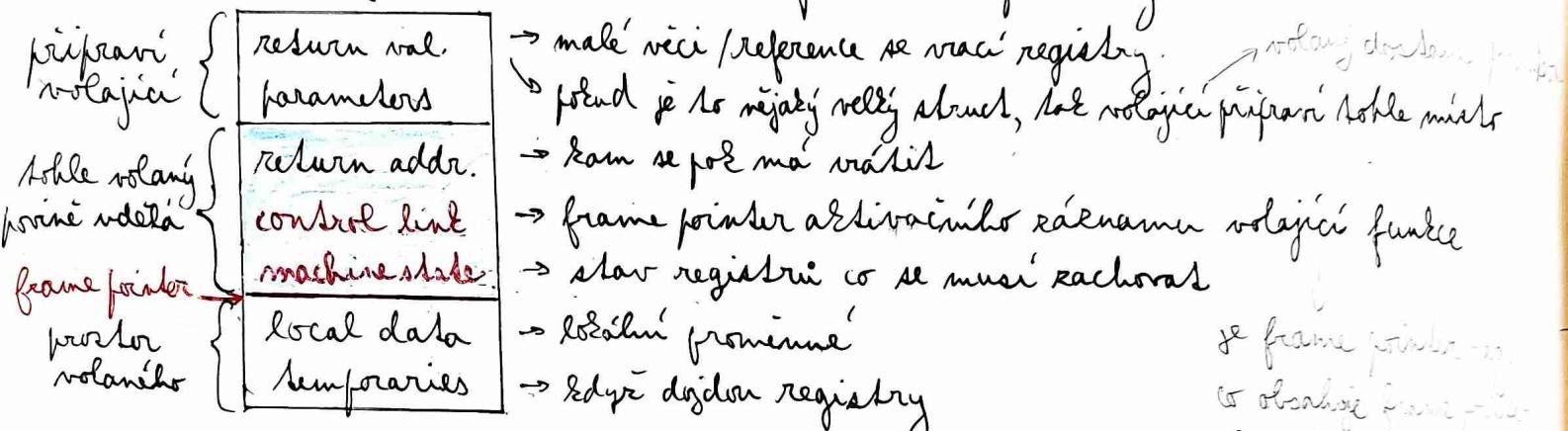
## Běhová podpora jazyka = Run-time

- statická - compiler + realizace rozhraní Environ
- dynamická - konstruktory a destruktory globálních objektů  
organizace paměti + obsah paměti před spuštěním  
volání konverce funkcí  
dynamické Environy toho jazyka

## Volání funkce

stack frame

- funkce při zavolení dostane aktivaciční ráznam a v něm pracuje
- ↳ někam jinam nesabí - kromě globálních proměnných



- tento krok aby aktivaciční ráznam je na zásobníku

## Volací konvence

- řídí se jí všechny překladače, aby to bylo kompatibilní
- public name mangling → linker jen bloufá porovnává jména publik a externí  
"manglední jmén" <sup>→ externí / předěláno</sup> ⇒ aby fungovalo např. přetištění / více konstruktorů
- musí se domluvit kdo co uloží z toho zásobníku + call / return sekvence
- předávání parametrů → na stack / registry + v jakém pořadí se na zásobníku uloží
- místníková hodnota → bude registrama / přes stack
- registry - které se musí zachovat (preserve) a které ne (scratch)  
↳ jaké jsou role registrů

## Call / return sekvence = odpovědností při volání

- volající předává parametry a po návratu volané funkce je i smazí
- volaná funkce smazí svoje lokální data, obnoví machine state,  
nastavi frame pointer reg. na control link, vrátí se a smazí tohle všechno
- machine = posunutí pointrem na všechny zásobníky
- existují volací konvence, kde funkce volaná uloží i ty parametry

- Předávání parametrů
    - hodnotou - funkce kopíruje argument a využívá jeho lokální proměnné
    - referencí - předává se adresa, v C++ pomocí & → funkce mění lokální variabilu na daném stacku se dala do adresy → instrukce přesílájící referenci adresu
    - čistý C řeší předávání referencí tak, že hodnotou předá pointer
      - ↳ rozdíl: pointer nemá fixní + musíme explicitně psát `da *`, kde je to strava
  - Proměnné
    - = pojmenování bloku paměti, který obsahuje hodnotu
    - má nějaký typ (ve staticky typovaných jazycích)
    - kde leží?
  - C/C# Python
    - static data - globální proměnné → alokace při spuštění
    - stack - lokální proměnné → alokace na rámcovou paměť při běhu programu
    - heap - dynamické proměnné → alokace na heap pomocí `new (C#) / malloc (C)`
    - slovník - (klíč, hodnota) = (jméno proměnné, info o proměnné)
      - ↳ Abylo funkce funkce Python, javascript, ... → typ, hodnota, adresa, ...
  - Heap
    - dynamická paměť, kterou si při výpočtu musíme brát
    - není to ten stack, takže se to při návratu z funkce změní jako lokální proměnné
    - funkce volaná může něco dynamicky alokovat a vrátit fci volajícímu pointer
    - alokace - evidenční rolník bloků, alokační algoritmy
      - je to explicitní → např. `new List <>` → Abylo vrácení pointeru
    - dealokace - C, C++ jde explicitně užít `free / delete`
      - ↳ zavrhování se na to + můžete se snažit exceptions → na heapu Abylo nezvládat
      - ↳ memory leak = nemáme GC a ztrácíme všechny pointery na tu paměť
  - Garbage collector = automatická dealokace, docela formálne algoritmy → mega fast alokace
    - GC se nekontroluje všechny paměti → nemusíme řešit allokacní algoritmy
    - GC je formálý + není možné alokovat když běží → freeze
    - hodné programů spoléhá na to, že se GC nikdy nekontroluje → podtržuje fragmentaci
    - heap consolidation = po tom co GC dojde, soubory aktivní bloky, sesypnou na stranu
    - rotování = rožkon, může dojít k závratě v nepředvídatelný moment
  - GC strategie - rozhodne se používají posledních
    - trasování → projdě aktivní ráckamy a najde živé proměnné → k nim to jde dále rekurenci tracing → nazonec snadě všechno co nemá oznámeno jako živé
    - posítání referencí → když uděláme novou lokální referenci → ++count
      - ↳ když ta lokální proměnná s touto referencí zanikne → --count
      - když count = 0, tak uděláme free
      - pozor na cykly / odlehle komponenty souvislosti - trasování. Tento problém nemá

## Přenositelnost kódu

- aby to fungovalo na více OS, překladačích, architekturách, ...

### 1) CPU architektura

- C, C++ → různé velikosti typů, kód je podle architektury → ab je možné si využít, že senklu má 32 bit
- C#, Java → fixed velikosti typů → int je vždy 32 bit
- někdy je kód řešit i endianitu dat

### 2) překladač → gcc, msvc, clang

- C++ má více překladačů s různými speciálními funkcionality
- ⇒ řešení: používá syntaxi a tukhorny ze standardního jazyka
- překladače se snaží dodržovat ISO standard toho jazyka
- kód C# nebo Java májí jen 1 překladač ⇒ ten problém tam nemá

### 3) OS

#### systémové volání

- různé sys-cally, ale funkce to má stejné → OPEN, ...
- jen různá jména a interface (někdy to může vyjídat zbytečně)
- podmíněný překlad = ve zdrojáku označím co se má přeložit podle OS
- Java a C# obě všechno dělaj sami za runtime ⇒ je to samotně fajn

## Přenositelnost pomocí Virtual Machine

Java: byte code  
p C#: IL

- Java, C# se přeloží do instrukcí procesoru co neexistuje ⇒ metód
- ten metód dostane nějaký nativní VM, co je za runtime interpretuje
- ↑ bezpečnost a kontroly → roví se to do sandboxu a ten interpret kontroluje, jestli to nedělá něco co nemá
- řešení problému rychlosti
- Just-in-Time = JIT → jednokrát funkce, binary, ...  
→ postupně za běhu se metód překládá na nativní kód a to se běží nadefinované runtime
- ⇒ nemusím to překládat ohnáštět - Edgě volám poprvé nějakou funkci, tak se běží podruhé už se zkontroluje překladač
- Ahead-of-Time = AOT  
→ přeloží se to celé při instalaci a spustí se to už přeložené

## • Operacní systémy

→ nemá přesná definice, ale má 2 hlavní úkoly

1, abstraktní stroj = abstrakce nad HW

→ reprezentování Kernel API → systémové volání (jsou schovány v nějaké knihovně)

→ schovává před programátorem tu HW implementaci a poskytuje to C rozhraní

2, resource manager = přidělování prostředků programů

→管uje všechny HW a přiděluje aplikacím HW prostředky

→ Ty aplikace k tomu musí mít sdílet mezi sebou

→ alokace (mem), time-sharing (CPU), abstrakce (disk, síť, grafika ...)

## • Režim procesoru

1) uživatelský mód → pro všechny aplikace, nemá přístup k nějakým reg. + instrukcím

2) kernel mód → používá ho OS nebo jen nějaká jeho část - chybek běží v user

↪ kernel = ta část OS co běží v kernel módu

• přechod kernel → user : OS k tomu dělá triviálně - jde o special register

• přechod user → kernel → např. některé sys-cally potřebují kernel režim  
entry pointy do kernel režimu = adresy jak k tomu přepnout

↪ jsou jasné definovány a OS je kontroluje → třeba na to je special instrukce

↪ po přepnutí ta aplikace skočí na nějaké jasné definované místo v paměti, kde pro ni OS připravil ty instrukce

↪ parametry pro ten sys-call (jméno souboru) se předávají registry

→ entry point může být i tak, že se ten program pouze udělá z nějakou systémovou instrukcí, což CPU pozná když je chyba a předá řízení OS

↪ Ta sys instrukce je nějak definována pro ten přechod

to bývá napsáno v C

OPEN  
PRINTF

## • Architektura OS

- monolitická → zde se dělaly funkce OS → hromadné kódování
    - všechny globální věci, která celá běží privilegovaně
    - nemá to formálnou strukturu - kromě entry-pointu, re kterého je nejedna volba servisních procedur co ještě vedou dále na utility procedury
  - ⊕ velmi efektivní, hodně rychlé sys-cally
  - ⊖ však všechny programy prostě budou chybky + když se někdo dostane do OS, tak může dělat co chce a má přístup k celému počítací
  - ⊖ původně se to ani nedalo rozšířit, dnes je možné načítat dynamické moduly ⇒ je tam ještě větší prostor pro chybky

- restornata → evoluce těch starých monolitů

- organizace OS do vrstev s řízenými službami a oprávněními
  - vrstva  $m+1$  může využívat funkce služby vrstvy  $m$
  - ⊕ bezpečnější a jednodušší rozšířitelné
  - ⊖ nároč rozhramí mezi vrstvami je velmi náročný

- microkernel

- snaha je udělat kernel co nejménší → třeba méní mezi 1MB
  - nad jádrem jsou jednotlivé moduly (filesystem, rádič disků, ...)
  - kernel registruje komunikaci mezi tyto moduly
  - posílání správ jde mezi klientem a serverem
    - nezávisí na celém OS
  - když nějaký modul spodne, totž ho kernel rázce spustí
  - kernel i ty moduly mají svoji protected paměť, do které si nevidí
  - (+) bezpečné, spolehlivé, snadno rozšiřitelné
  - (-) je to hodně formát - kvůli posílání těch správ
  - windows se snaží být mikrokernel
  - když mikrokernel má ani nezávisí na architekturě CPU, protože ještě pod ním je Hardware Abstraction Layer

- Záručení = HW důvodem složití je nejistotu výkazu

- device controller = radic  $\rightarrow$  radice di un esponente

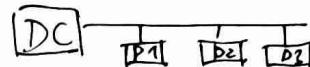
## Topologie zarizení

DC = Device Controller

- sdílená sítěnice → hlavní disk

→ musí se řešit arbitraci pro hodné zarizení

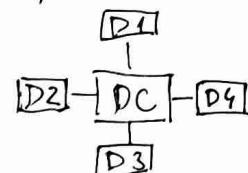
→ průtok dat je omezený ⇒ pro hodné zarizení řasné



- star → dnes u disků

→ zarizení je k pripojené & řadici

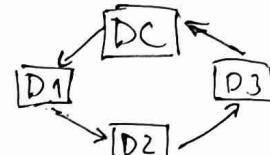
→ nevýhoda je složitější konstrukce řadice



- ring → moje se možností má

→ orientovaná sítěnice se zarizení

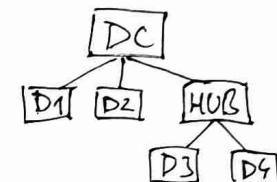
→ levnější řešení, vellá latence



- tree

→ DC je v kořeni, může se větvit formou hubu

→ řadici i vlastič musí být schopni svádat změny topologie zohlednu



## Komunikace se zarizením = Device Handling

→ abyste řekli OS, některá žádost OS je nezávislá na vlastičích a řadicích

→ programátor může udělat přímo sys-call, ale pro přednost 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

Samotaj  
převodovka

→ ta read žádost se dostane až k vlastiči

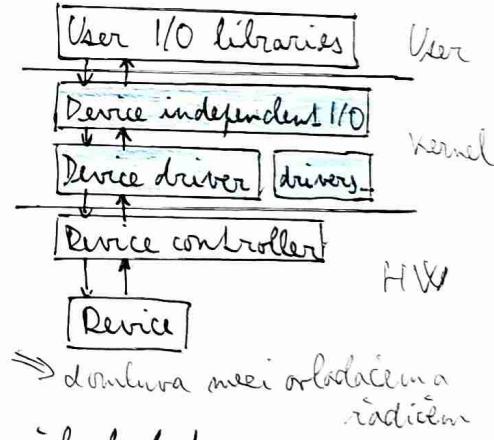
→ vlastič dá request řadici zohlednu disku

čitání } → ta HW komunikace mezi řadicem a diskem

může probíhat různým způsobem, např. přes výzvání

→ až budou data ready, tak dá vlastič řadici standardní emulativní adresu na ni → ta ji dá mě

→ problematická je ta část, kdy vlastič čeká na ty data



- polling

→ CPU periodicky kontrolouje stav zohlednu řadici a čeká až bude hotovo

→ musíme aktivně čekat + to má latenci kvůli periodicitě

- interrupt

→ zarizení posle CPU signál, že má hotovo ⇒ CPU přeruší aktuální

pravidlo instrukcí a sloučí do obsluhy přerušení - ten to vyzvá a vrátí se

→ řadici je držena připojeny & CPU pro ten signál ⇒ CPU na to potřebuje pin

- DMA = Direct Memory Access

→ DMA řadici dokáže přenout ty vypočítané data z řadice zohlednu řadici do paměti

⇒ až budou data v paměti, tak se vyvolá interrupt

→ scatter/gather → DMA dokáže z nesouvislých dat vybavit 1 blok nebo

naopak ten blok rozsekat → když jedna paket, tak ho rozseč na

TCP header, IP header a ty aktuální data

PCI je na výběr  
gige net card

PCI je na výběr  
gige net card

CPU nemusí mít řadiče ⇒ řadič se využívá

nebo opakně do zarizení

- Typy přerušení = interrupt
  - externí → z venku přes nějaký pin na procesor
    - ↳ je možné ho zamaskovat = maskávat / povolit → v kernel módu
  - exception = výjimka
    - neocitávané vyvolaná nějakou spolurom instrukcí → CPU ji vyvolá sám
    - u všech těch výjimek je definováno co se má stát když nastane
      - ⇒ střeba se skočí na nějakou jinou adresu
    - využití pro emulaci nových instrukcí na starých procesorech
      - ↳ když přijde ta instrukce, tak nelam skočinu, spočítáme to a vrátíme se zpět
  - fault = ta instrukce nemá schopnost dohlížet
    - udělá se rollback a to přerušení se udělá před touto instrukcí
  - trap = ta instrukce se provede a až poté nastane to přerušení
- softwarové
  - speciální instrukce co vyvolá přerušení ⇒ skočí do OS
  - slouží jako entry-point OS pro sys-cally
  - pro ty přerušení je reálně nějaký řádící, který CPU posílá číslo toho přerušení formou nějakého protokolu ⇒ CPU poté skočí do obsluhy přerušení
  - adresy těch obsluh jsou buď fiktní nebo v interrupt table
- Processing → OS má pouze jednu programovou aktivitu procesu
  - program → spustím → proces → vloženo = stream instrukcí do procesu
    - ↳ program → sys-call → PID
- process = instance programu, je to nějaký objekt OS
  - vykročí do OS pomocí sys-callu.
  - OS mu poskytuje prostředky a dělá si jejich evidenci, aby si je mohl vrátit spásky ať už proces skončí
- vložka = thread
  - 1 proces může mít více vložek a běží paralelně
  - je to základní jednotka kernel schedulingu
  - kontext procesoru = DS pro ukládání stavu vložky
    - ↳ pamatuje si starý registr → je tam i stack pointer
    - když CPU připina vložku, tak musí udělat context-switch
- fiber
  - ještě lehčí jednotka plánování než vložka
  - když má svůj kontext a souborník
  - umožňuje kooperativní plánování
    - ↳ fiber 1 řeší „já už nechci běžet“ a vybere další fiber

## parent-child processy

- Línix
- proces může spustit další proces a protože má jeho PID, tak počítá až do téhle
  - dítě si pamatuje PID rodiče → PPID → když skončí, tak mu posle signal
  - killnou procesu nekillne jeho dítě ⇒ adoptuje je init proces PID1
  - ⇒ zabít abron procesů je celém systému

• Scheduler = část OS co přiděluje výpočetní zdroje vláknům  $\Rightarrow$  plánovací algoritmus

• Multitasking - na 1 jádru běží více procesů současně  $\Rightarrow$  je tam nějaké přednostní

• Multiprocessing - mám více procesorů / jader  $\Rightarrow$  pro ten scheduler je to náročnější  
→ affinita procesu = snaha aby tohle vlácko běželo před na stejném jádru  
→ kvůli tomu že máme více jáder

• Real-time scheduling  $\rightarrow$  jiná verze schedulingu pro nestandardní situace

↳ RT proces má začátek (release time) a konec (deadline)

$\Rightarrow$  musí do téhle doby v rámci intervalu  $\rightarrow$  třeba je mega složitý plánování

↳ hard deadline  $\rightarrow$  nemá cenu to dodělávat, když jsem to neslýchal

↳ soft deadline  $\rightarrow$  před  $\rightarrow$  má cenu dopočítat

$\Rightarrow$  třeba airbagy  $\vee$  autě nebo řízení jaderné elektrárny

## Stavový diagram plánování

→ vlácko = unit of scheduling = jednotka plánování

→ každé vlácko má svého stavu

• created  $\rightarrow$  čeká to na spuštění (release time)

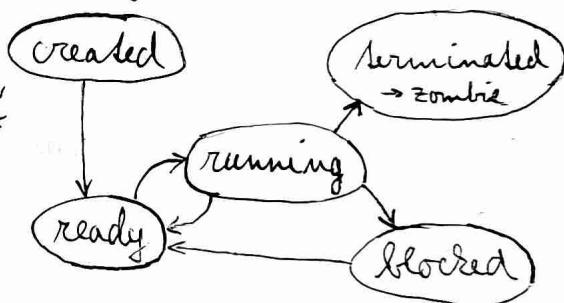
• ready  $\rightarrow$  schopný běžet, ale nemá přístup k CPU

• running  $\rightarrow$  počud už běží moc dlouho, tak zpět do ready

• blocked  $\rightarrow$  čeká na zdroje - třeba čtení ze souboru / user input zombie state

• terminated  $\rightarrow$  už skončil, ale rodičovský proces si ještě neříká výsledek

$\rightarrow$  v tomto stavu si OS bere zpět všechny prostředky



• Multitasking  $\rightarrow$  mám 1 jádro a chci na něm spuštět vlákná  $\rightarrow$  jak na to?

• kooperativní  $\rightarrow$  OS se o to nestará a všechny vlákná spolu spolupracují  
 $\rightarrow$  možné jenom ve speciálních systémech

## preemptivní

$\rightarrow$  každé vlácko dostane time slice (více 20 ms)

$\rightarrow$  OS má nějaký externí HW zdroj přerušení  $\rightarrow$  timer

$\rightarrow$  počud to vlácko během toho time slice neskončí nebo se nerabuje,  
tak na konci time slice dostane interrupt a jede do stavu ready

$\rightarrow$  OS udělá context-switch a pustí tam jiné vlácko

## Scheduling - cíle

- maximalizovat využití CPU a jader  $\rightarrow$  počet jader prošebují  $\rightarrow$  využití všechny jader
- spravidlivě rozdělit výpočetní zdroje mezi vlákena
- maximalizovat propustnost = # vláken co dokončí za jednotku času
- minimalizovat dobu jednoho vlákena
- minimalizovat čekací dobu v ready stavu
- minimalizovat čas na odpočívání v interaktivních aplikacích
  - $\hookrightarrow$  když sličnou možností, tak musí rychle změnit stav
  - blocked  $\rightarrow$  ready  $\rightarrow$  running a méně udeřit

## Priorita procesu

- = číslo co stanovuje důležitost procesu  $\rightarrow$  a vlákena mají prioritu většinou dlejší
- $\rightarrow$  priorita je součet statické a dynamické priority
- statická - přidělí se při spuštění procesu a následně se pouze snižovat
- dynamická - jednou za čas se ready vláknům incrementuje
  - $\hookrightarrow$  když přejde do stavu running, tak se využívají

## Plánovací algoritmy

- nepreemptivní  $\sim$  kooperativní  $\rightarrow$  strašně simple
- First Come First Served - fronta, vypočítám vláknou a jede další
- Shortest Job First - musím předem vědět horní odkaz execution time
  - $\hookrightarrow$  maximizuje propustnost = throughput
- Longest Job First
- preemptivní
  - Round Robin  $\rightarrow$  jde FCFS
    - $\rightarrow$  je tam prioritní fronta + každé vlákně má time slice
  - Multilevel feedback queue  $\rightarrow$  dynamický algoritmus
    - $\rightarrow$  několik front podle toho jak dlouhé timeslice by vlákena chtějí
    - $\rightarrow$  horní fronta má nejméně timeslice a spodní nejvíce
    - $\rightarrow$  pro výběr procházím fronty zeshora dolů dokud nenajdu neprázdnou frontu
    - $\rightarrow$  potom se vláknou zahlopuje před koncem timeslice tak ho dál výběr
    - $\rightarrow$  potom na konci timeslice pokračuje, tak mít
    - $\Rightarrow$  interaktivní procesy sloučí v nejvyšší vlastnosti  $\rightarrow$  většinu času jsou využívána
- Completely fair scheduler - CFS  $\rightarrow$  Linux scheduler
  - $\rightarrow$  stručně: vybírám proces co ještě nebyl moc dlouho a timeslice  $\sim$  doba čekání
  - $\rightarrow$  # proces si pamatuje svůj virtual runtime = jak celkově dlouho byl ve stavu running
  - $\rightarrow$  jádro má červeno - černý strom s ready procesy  $\rightarrow$  indexace podle VRuntime
  - $\rightarrow$  vybírám nejlevnejší uzel = proces s nejménším VRuntime
  - $\rightarrow$  timeslice = čas čekání / # procesů ve strome  $\rightarrow$  když je ready vláknem

## Komunikace mezi procesy

- OS schovává prostředky procesu před ostatními procesy (např. paměť)
- kooperující procesy si chcejí nejat bezpečně poslat správy
- pipes → něco jako sítinové sítě ⇒ na jedné straně dávám data a na druhé je bere
- sdílená paměť → dělá se to přes sys-calls
- signály → používá ji i kernel

z pohledu OS

## Soubory

- = data co spolu nejde souviset / abstraktní proud dat
- kernel OS nerozumí semantice těch dat

## Identifikace souboru

- filesystem používá číselné ID
- soubor má jméno a cestu & může sloužit různou stromovitou struktuře
- tohle tam je pro lidi + některé části jména mohou mít speciální význam a <sup>get</sup> (0,1,2) = (std.in, std.out, std.err)

## Operace

- open → najde soubor a načte si jeho metadata + vrátí file handle
    - ↳ handle je číslo, které kernel dodává pomocí tabulky přidružit na ID toho souboru
  - close → uloží změny, odstraní ten objekt metadata a vydá z tabulky file handle
  - read + write → používáme ten file handle, write se dělá do paměti a očekává se na disk
  - seek → dělá se abstrakce ukazoválka, ale seek se jenom lineárně posouvá
- ⇒ každý proces má vlastní tabulku otevřených souborů - je to koncept objektu proces

## create, truncate, delete

- flush = zapísání změn co jsou nalešovány v paměti na disk
  - ↳ volá se náhodně a při zavření souboru

## změna atribut

extension například OS/aplikaci co to je

adresa na adresu

## Atributy souboru → jméno, velikost, typ, práva, timestamps, ve kterém sektoru běží

## Buffering → předčítání souboru, protože to je slow

OS: nalešované sektory se nečítají dvakrát

C,C++: běhová podpora si toho sama řeší

serverní čtení → OS by sektory četla dopředu = read ahead

## Synchrozní I/O: read() zablokuje všechno, čteče, vráti → všechno ready

## Asynchrozní I/O: read() rozhání operaci ale hned se z toho syscallu vráti

⇒ všechno běží dál a OS současně čte z toho souboru

→ pokud ho nepřečte vrás, pak se to všechno dočasně zablokuje

## • Adresář = sada souborů

- většinou reprezentovaný jako soubor s nejatým special typem
- hlavně je to user-friendly a formátován při hledání open()
- někdy si pamatuje nějaké atributy souborů v něm → podle filesystemu
- je nějaká hierarchie s rootem
- operace: create, delete, rename file/subdir, search for name, list members

## • Uložisko souborů

- na disku (externí paměť), RAM, v síti → musí tam být nějaký filesystem
- virtuální soubory → OS pamoci nich poskytuje nějaké funkce navíc /dev/null

## • File links

- link (hard link) - více poloh v FS ukazuje na stejná data → stejně abs. file ID  
→ většina operací je transparentních, ale občas je to problematické
- symlink (soft link) - neštáť jehož obsah je cesta k jinému souboru  
→ je potřeba explicitně říct „follow symlinks“

## • File system = datová struktura v uložisku

- řeší jak a kde jsou data uložena + poskytuje abstrakci souborů pro OS
- musí umět:

- 1) překládat jména souborů na file ID  
→ rozkládá se po cestu a rekursivně se volá na podadresáře ⇒ slow
- 2) pamatovat si lokaci dat souboru = sekvence bloků
- 3) management volných bloků  
→ spojár, bitmapa, ...

$$\text{blok} = \text{několik souvisejících sektoru}$$
$$4 \text{ kB} = 8 \cdot 512 \text{ B}$$

- ⇒ pravidlý soubor zabírá 1 blok = 4 kB
- locální FS → na disku → FAT, NTFS, ext 2/3/4, XFS, BTRFS
  - sítový FS → protokol pro přístup k souborům přes síť → NFS, Samba
  - disk je možné rozdělit do partitions - shodně tam musí být různé FS
  - na každému disku je partition table → velikosti oddílu + FS tam

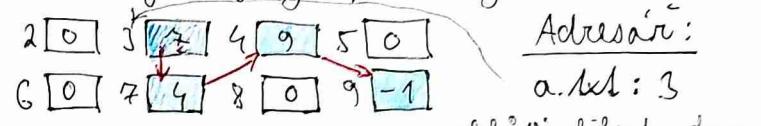
## • RAID = Redundant Array of Inexpensive Disks

- způsob jak spojit více disků do jednoho → HW nebo k dělá OS
- pro všechna data je potřeba (zakrývat diskové pole)
- je to rychlé a spolehlivé

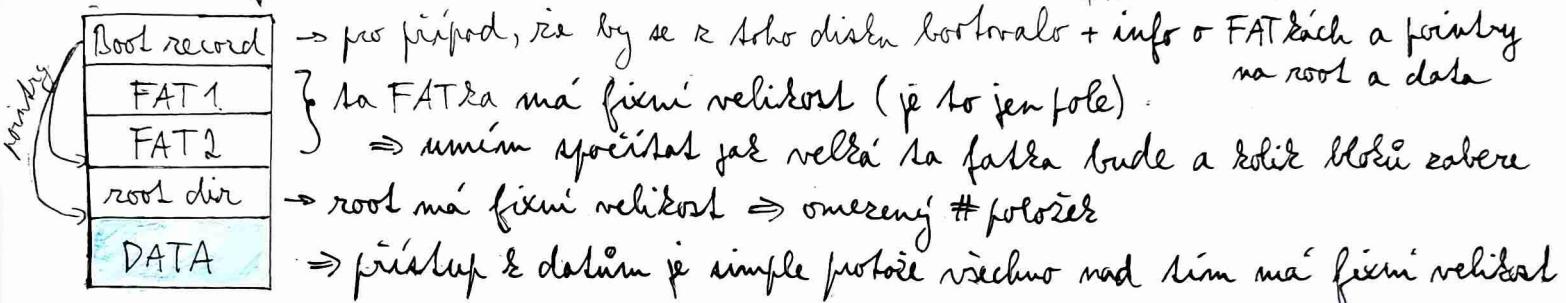
HDD disků

## FAT = File Allocation Table

- hodně simple, z dob MS-DOSu
- je tam struktura FAT co se stará o volné bloky a pozici souboru
- na tom disku jsou 2 FATy (kopie)  $\Rightarrow$  ráloha když počítal správné při zápisu
- adresář je speciální soubor  $\rightarrow$  na původním FATu 32B  $\rightarrow$  "disaster recovery"
- v něm je sekvence položek fixní velikosti s atributy
- pro každou položku si pamatuje číslo prvního bloku  $\Rightarrow$  ve FATce je spoják
- FATka je vlastně pole indexované čísly bloků a  $FAT[\text{block } n] = \text{block } n+1$
- ⇒ to pole začíná indexem 2 (0 a 1 mají nějaký special význam)
- 0 = prázdný blok
- 1 = poslední blok souboru



→ na disku je:

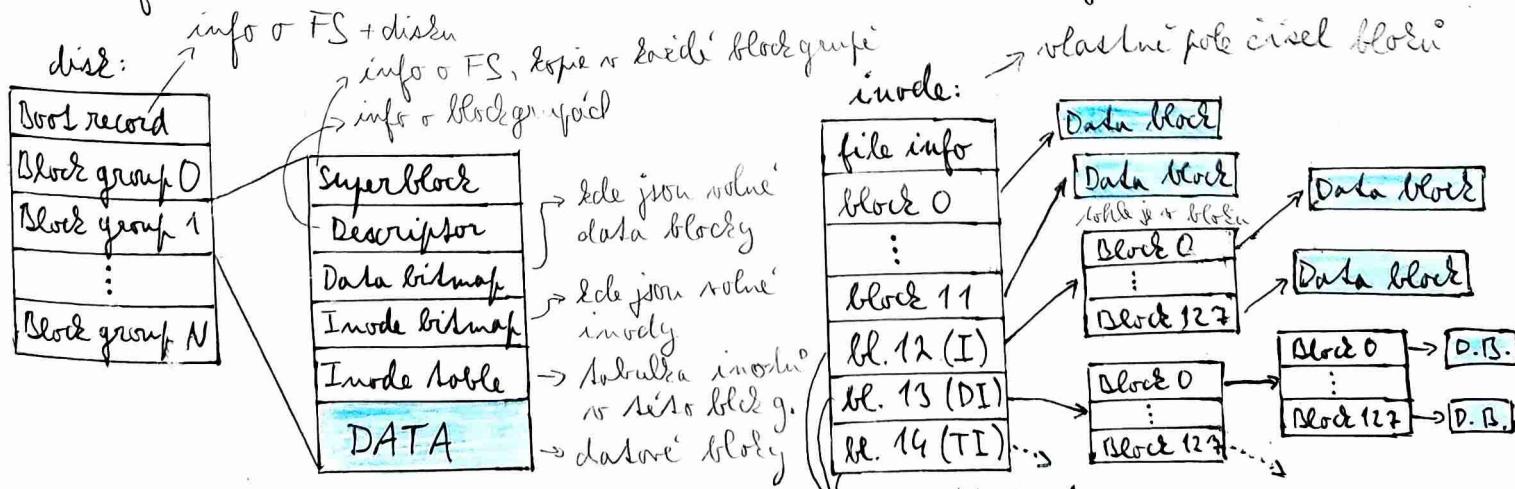


⊖ Open je slow protože při prohledávání adresáře ho musím lineárně projít

⊖ ten spoják je jednosměrný  $\Rightarrow$  nemůžu počítat seky dozadu

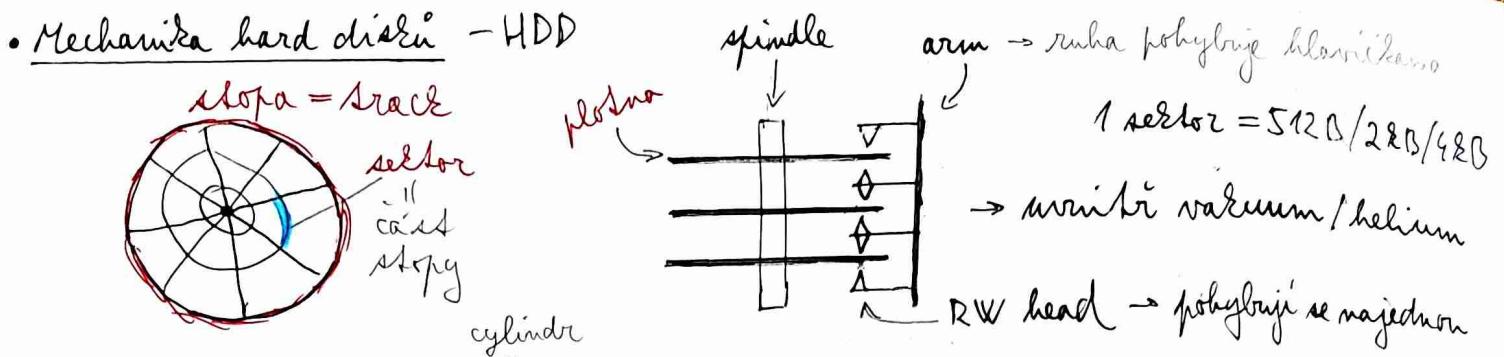
## • ext2 - původní Linux a bylo celkem simple

- ext3 přidal journals aby se zlepšila disaster recovery
- ext4 umožňuje uladit větší individuální soubory
- používá index nodes = inode  $\rightarrow$  1 inode reprezentuje 1 soubor / adresář
- adresář je rase speciální soubor ve kterém je sekvence položek
- s fixní strukturou  $\rightarrow$  1 položka = inode number, file name



→ nahodný přístup do souboru je OK

→ max. velikost souboru je  $(12 + 128 + 128^2 + 128^3) \cdot 4 \text{ KB} \approx 8 \text{ GB}$



cluster = stejná stopa na všech plátnách

blok = stejný sektor na všech plátnách

→ 1 blok používá FS

- flying height = vzdálenost blávy od plátny  $\sim 5\text{ nm}$

- rotací rychlosť = 5400, 7200, 10k, 15k rpm

→ bottleneck je mechanický pohyb blávy

→ složitá indexace sektoru  $\Rightarrow$  používá se LBA = Linear Block Addressing

## Disk scheduling

$\rightarrow$  musíme rozhodnout pořadí vykonávání RW requestů

$\rightarrow$  důležitou roli hraje OS, dnes k tomu dělá disk

access time = seek time + čas na rotaci + čas na transfer dat

$\rightarrow$  cíl je minimalizovat access time pro více I/O requestů

- FCFS = First Come First Served  $\rightarrow$  pro malou ráží  $\Rightarrow$  neefektivní

- SSTF = Shortest Seek Time First  $\rightarrow$  starvation = vzdálené requesty se nesplňují

- SCAN = algoritmus výstahu  $\rightarrow$  pro širokou ráží

- CSCAN = Circular scan  $\rightarrow$  dojede až k konci a pak se vrátí do prvního a zase dolů nenechá pasážery  $\rightarrow$  repeat

- LOOK / CLOOK - jako SCAN/CSCAN ale menovitěji končí diskem  $\rightarrow$  pokračuje

- FSCAN - 2 fronty, algoritmus pracuje s 1. a nové requesty dává do 2.

## Solid State Disk - SSD

- bez pohyblivých částí, rychle se odebírávají  $\rightarrow$  blávě zápis a macání

- organizace do bloků rozdělených na stránky  $\rightarrow$  1 page  $\sim 2-16\text{ kB}$

- read/write je po stránkách

- erase je po blokách  $\rightarrow$  ty stránky se označí jako invalid

- $\Rightarrow$  když se rozbití 1 stránka, tak přijde o celý blok

- dělá se GC s data consolidation = odstraní se invalidní bloky

- pomocí special FS co to tak rychle mení & validní stránky se seřazou na stránky

- 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 do hardware → (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 kódém přístupu do RAM → převod
  - proč to děláme?
    - hodi se větší adresový prostor (VAS může být větší než PAS)
    - převýšenou 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

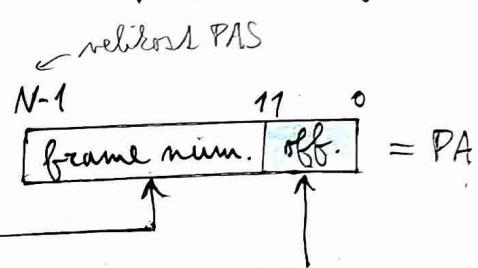
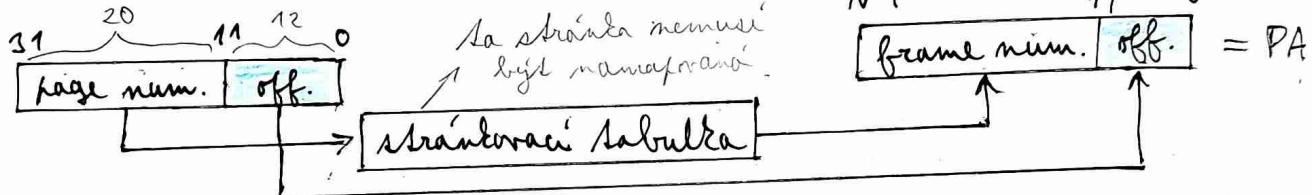
## segmentace

- koncepty:
  - VAP je rozdělený na logické segmenty - stejně jako paměť běžícího procesu
    - segmenty jsou nějak označovány - třeba má vlastní čísla (číhají se od 1)
  - virtuální adresa má 2 části → číslo segmentu, offset v rámci segmentu
    - by instrukce s tím musí pracovat
  - operační systém spravuje segmentační tabulku - pro každý proces
    - je to pole indexované segmenty obsahující deskriptory těch segmentů
    - v deskriptoru je barvá fyzická adresa toho segmentu + délka + atributy
    - OS jenom musí zajistit aby když běží někde proces byl někde
      - registr pointer na tu segmentační tabulku ⇒ CPU v ní kde leží
    - $PA = ST[\text{Segment num.}] + offset \rightarrow$  pokud  $offset \geq \text{délka segmentu}$  ⇒
      - pokud je číslo segmentu out of range ⇒ vyvolá se výjimka segment fault
      - když se kontroluje jestli bude někam psát do read only segmentu, ...
  - co když dojde prostor ve fyzické paměti?
    - nejedná se o celý proces na chvíli → v deskriptoru je present bit
    - pokud přistupují do segmentu co nemá present ⇒ segment fault
    - OS ten segment načte z disku a jede se znova
    - výpadek segmentu
  - spolu se řízením managementem fyzické paměti → méně fragmentace
    - je třeba hlavní důvod proč vše se segmentací neponává

## Stránkování

- VAS je rozdělený na stejně velké části (stránky), velikostí  $2^m$ 
  - ↳ velikost VAS je taky  $2^N$  → pro 32/64-bit CPU  $N=32/64$  → většinou
- PAS je rozdělený na stejně velké rámečky = frame
  - rámeček májí stejnou velikost jako stránky → většinou  $4KB = 2^{12} B$
- virtuální adresa je 1 číslo → narození od segmentace
- stránkovací tabulka - pole v paměti pro  $\forall$  proces
  - indexová číslom stránky → obsahuje číslo rámečku + atributy
  - v těch atributech je case příznak  $P=$  present, v rámci jestli mapování existuje
  - pokud mapování neexistuje / jiný problém ⇒ page fault = výfodek stránky
  - pro 32 bit VAP a 4KB stránky:

VA:



- tiské funguje protože stránky a rámečky jsou stejně velké a  $2^m$
- VAS a PAS můžou být různě velké

- 1 logický segment v tom procesu je rozdělený do více souvisejících stránek ale ty rámečky nemusí být ve fyzické paměti souvisle za sebou
  - ⇒ když dojde paměti tak nemusím vyhovorovat celý segment ale jen 1 stránku

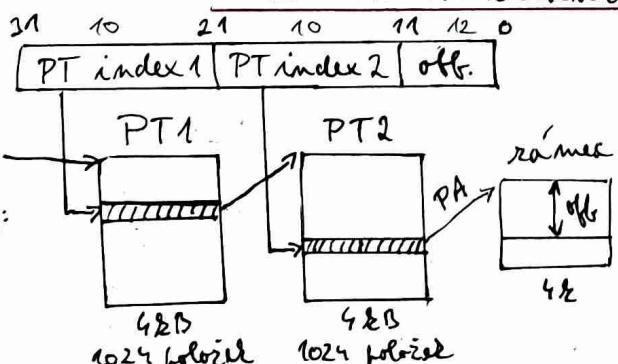
## Problemy stránkování

1. velikost - pro 1-úrovňovou tabulkou ↑ to je 1 pole

↳ pro 32-bit VAS je  $2^{20}$  4KB stránek ⇒  $2^{20} = 1M$  položek

→ pokud 1 položka = 4B, tak si musím pro  $\forall$  proces paměťovat 4MB tabulek

⇒ řešení: víceúrovňová stránkovací tabulka → ∵ nepotřebuju celý VAP



- v PT1 jsou fyzické adresy PT2
- v PT2 jsou už fyzické adresy rámečků
- ta tabulka má 4KB = 1 stránka
- PT1 musím mít všechny v paměti
- by PT2 mapují podle potřeby

- rámečky v PT1 mají pořad present bit ⇒ může být page fault na PT1
- 1 tabulka 2. úrovně stačí na adresování  $1024 \cdot 4KB = 4MB$  paměti

## 2, rychlosť

- priečinok na PA vyzaduje viac prístupov do pamäti  $\Rightarrow$  to je mega slow
- $\Rightarrow$  nakoľo užívame to do TLB = Translation Lookaside Buffer
  - TLB využíva associatívnu pamäť  $\xrightarrow{\text{vz. tabuľka m. adresami}}$  slovko (stránka, rámc)  $\xrightarrow{\text{vz. tabuľka m. adresami}}$
  - využívanie toho, že procesy často prístupujú k stejnej stránke

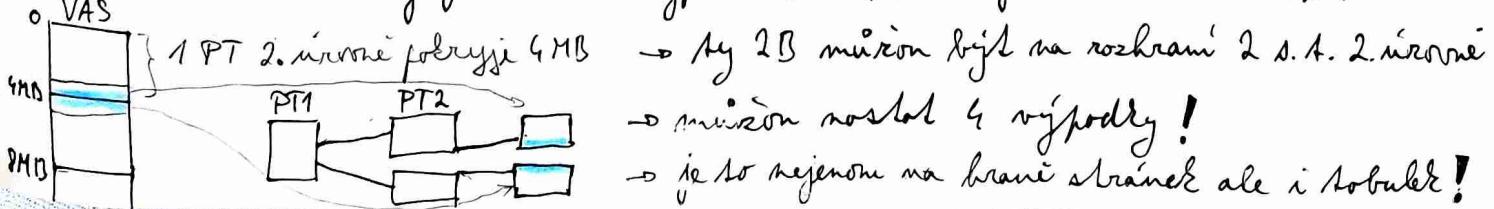
## • Algoritmus pre prenos adresy u stránkovania $\rightarrow$ HW

1. rozdeľ VA na číslo stránky a offset
2. checni jestli to už nemá v TLB  $\rightarrow$  pokud tam je toto konkrém
3. použi stránkovaciu tabuľku
  - rozdeľ číslo stránky na kolik časťí kolik je úrovňi
  - jdi do PT1  $\rightarrow$  pokud tam je adresa PT2 jdu dál  $\xrightarrow{\text{pouze present bitka}}$ 
    - pokud tam adresa dolnej tabuľky nemá  $\Rightarrow$  page fault
    - z poslednej úrovne vyzvedni číslo rámcu a ulož ho do TLB
4. aktualizuj príčiny A = Accessed a D = Dirty v stránkovacej tabuľke a TLB
  - $\rightarrow$  A ~ stránku nikdo použil, D ~ do stránky nikdo napsal
  - $\rightarrow$  musím je zmieniť i keďže jsem se k tej stránke dosiahol pres TLB
5. riškej fyzickej adresy k číslu rámcu a offsetu

## • Tak se rieši výpadek stránky?

- OS interrupt handler musí zjistiť kde sa stala chyba u toho stránkovania
- ktorý proces běží  $\Rightarrow$  v objektu procesu je pointer na tu stránkovaciu tabuľku
- možné dôvody:
  - neoprávněný zápis, čižní se systémové pamäti, ...
  - pokus o prístup do ďalšej nenaoborovanej virtuálnej pamäti
  - vše OK ale neexistuje mapovanie
- Jak vytvoriť mapovanie?
  - OS najde volný rámeček nebo vybere obecť algoritmom na výberu stránky
  - pokud je obecť dirky (zmienila sa) tak ten rámeček uloží na disk
  - musím skontrolovať mapovanie obecť v TLB  $\xrightarrow{\text{TLB nemusí byť následne, ale vtedy HW}}$
  - na volný rámeček nahráju svoj obsah a opravím stránkovaciu tabuľku
  - vrátim sa späť do handlu a skusím tu instrukciu znova

## • Vložka: Kolik miere najviac možno vypodeti stránku keďže máci 2B?



## Algoritmy na výměnu stránek

- využívají se jenétož situaci kde je potřeba vybrat oběť aby uvolnila místo  
⇒ vybírání rámců, cache, TLB

## Optimal page algorithm

- vybere stránku co bude nejdéle nefoučitá ⇒ nejménší page-fault rate
- jen teoretický, jeho snaha se lomuji přiblížit

## Hodiny (clock)

- rámců si sorganizuj do kruhu a dám tam rucičku - ukazuje na rámců
- využívá Access bit, který MMU nastaví na 1 každý na tu stránku sáhne
- rámců vybírám tak, že postupně rucička odkola:

  1. if  $A_{\text{rámců}} \neq 0$ :  $A \leftarrow 0$  a jdu dál ← rámců mlužu
  2. if  $A_{\text{rámců}} = 0$ : vyberu tenké rámců ← dleto vždyj používá

## NRU = Not Recently Used

- příznaky A se mluží pravidelně - třeba 1 za minutu
- rámců rozděl do 4 tříd používání podle příznaků A a D

A	D	třída	
0	0	0	→ nefoučitá + nezmíněné
0	1	1	→ nefoučitá ale musím ho rozdat na čist
1	0	2	→ někdo ho používá
1	1	3	

⇒ vyberu náhodný rámců z nejvíce nepravidelné třídy

## LRU = Least Recently Used → tohle se reálně používá

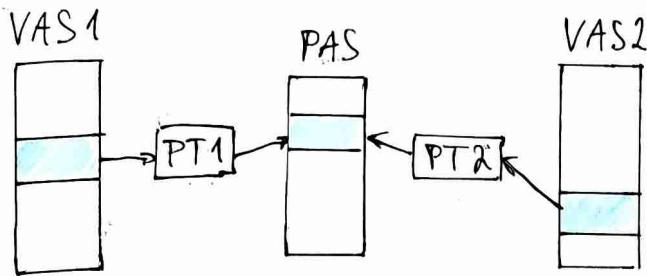
- používám minulost pro předvídanou budoucnost
- ⇒ vybere stránku na kterou nejdelsí doba někdo nesáhl
- existují HW implementace
- SW implementace:
  - zásobník - když na něco sáhnu tak to dám nahoru a oběť je na druhém
  - tohle je moc pomalé ⇒ aprotimace NFU

## NFU = Not Frequently Used - aprotimace LRU

- když rámců má počítadlo → někde bobem, do s. 1. by se nevěšlo
- pravidelně mlužu A a pokud  $A=1$  tak počítadlo incrementuju
- ⇒ vybere se rámců s nejménším počítadlem → aby se hned mohlo
- novým stránkám musím na začátku dát nějaké skóre
- staré framy by se nevybraly ⇒ aging: periodicky count  $\leftarrow \text{count} / 2$

## Sdílená paměť

- více procesů spolu sdílí část virtuální paměti - způsob komunikace
- někdo udělá sys-call „vyrob sdílenou paměť“ a další procesy se připojí

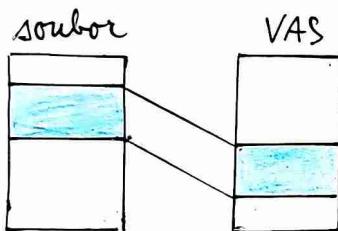


→ každý proces má svou s.č.  
a ta sdílená paměť je v VAP jinde  
ale ty rámců jsou sdílené  
⇒ ve sdílené paměti nemůže  
posuvat rámce → jenom offsety

- při 2-level stránkování lze mít společnou tabulkou 2. úrovně

## Memory-mapped files = paměťově mapované soubory

- normálně udeľáme open a pak dešlám read a write  
↳ ten soubor se mi naskládá někam do paměti ale pořád musím dešlat R/W



→ soubor si namapují do virtuální paměti  
⇒ do koho souboru můžu přistupovat  
operacemi procesoru load a store  
⇒ normálně nastavují A a D bity

- když se ta stránka vybere jako obecn., tak pojde D ≠ 0, tak užívám

## problemy:

- appendování na konec souboru ⇒ jsem tam nemapaovaný stránky
- soubor může být větší než VAS

## Virtual machine

- pěknější výraz, než 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 problémů se to musí postarat ten host OS - ale většinou se to nedělá
- normální instrukce běží na CPU, ale special kernel  
instrukce (nastavováním stránkování) emuluje ten host OS

⊕ isolace VM od reálného systému

⊕ encapsulace → VM ~ soubory ⇒ můžu dešlat snapshotsy když by to spadlo

⊕ kompatibilita → snadno se přenáší na jiný HW - VM posuzuje virtuální HW

↳ virtuální HW → virtuální můžou mít vlastní RAM, focič jader

- Kontejnery - virtualizace na úrovni OS
  - sjednodušená virtualizace → používá se reálný HW ne virtualní
  - OS kernel musí umožňovat existenci více oddělených user space
    - ↳ na kterém si v kontejneru nejdouším windows

## Paralelní programování a synchronizace

- Paralelní počítání = používá se více jader a instrukce se provádějí současně
- Concurrent počítání = multitasking na 1 jádru

### Problemy paralelního počítání

#### Race condition

→ když více vláken současně mění stejná data

⇒ výsledek se může lišit podle časování / schedulingu OS

⇒ není to deterministické

→ příklad se spojáním: LL l;

$$\begin{array}{ll} 1. \text{l. add}(A) \rightarrow A.\text{next} = \text{head}; & \\ 2. \text{l. add}(B) \quad \text{head} = A; & \end{array}$$

⇒  $a \in \{11, 12, 13\}$

řídí zápis do sloupu  
inter. pl

$$\left\{ \begin{array}{l} a=10 \\ 1. a=a+1 \\ 2. a=a+2 \end{array} \right.$$

#### Kritická sece

→ identifikují kritickou sekci kódu kde může vzniknout race condition

⇒ řešení = mutual exclusion ⇒ v k.s. 1 vláško at a time

#### Synchronizace - jak udelat mutual exclusion? = zachování integritetu

→ buď se kritická sece samočárá nebo se vláška nějak řídí

→ realizace pomocí synchronizačních primitive

• aktivní - žerou ias procesoru (aktivní čekání)

• pasivní (blokující) - kernel to vlášku zablokuje abud nemí přístup pro další

↳ pasivní se nemusí vždy vyplatit : sys-call na blokování je obvyklý

⇒ pokud je race condition vznášá, když je aktivní lepší

↳ (O-W) inst.

→ aktivní potřebují HW podporu

→ používají atomické instrukce → Test-and-set (TSL), Compare-and-swap (CAS)

↳ instrukce co CPU záručeně udělá celou a bude přeměněna

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

bool CAS: porovná rámeček s danou hodnotou a pokud jsou stejné

↑  
tak ho nastaví na novou hodnotu, vrátí true/false

## Spin-lock - aktivní

→ když rámeček = 0 tak je volno

⇒ všechni se ho pomocí CAS snáší nastaví na 1

⇒ avšak to však může dojít k kritickou sekci tak rámeček nastaví na 0

→ vhodné pro krátké čekání

## Semafor - blokující

→ fórmátlo a fronta čekajících vláken

→ atomické operace UP a DOWN ale je to sys-call → pravidly se to třeba pomocí spin-locku

→ fórmátlo se na začátek nastaví na # vláken co chceme najednou poslat do kritické sekce

→ když vlákno chce do kritické sekce, tak zavola DOWN

→ když s ním odchází tak zavola UP

down: ↗ je volné místo

if count > 0 : --count;

else: ↗ nemá volno ⇒ jdou do fronty  
queue.push (toto vlákno)

vlákno.block()

up: ↗ mávolné místo

++count;

if ! queue.empty():

v ← queue.pop()

v.unblock() ← odblokuje ho

--count; ← jede tam někdo místo mi

## 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 poté se ji odblokuje

→ je to dobré na synchronizaci vláken - vím, že všechny jsou tady

## Monitor - v programovacích jazycích

→ metody objektu nejsou triviální mají rámeček 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

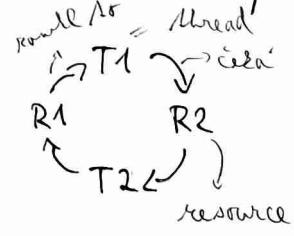
### Coffmanovy podmínky:

1. mutual exclusion: prostředky nelze sdílet

2. hold and wait: vlákno drží prostředek & čeká další

3. no preemption: prostředek nemůže vláknu jiného zabrat

4. circular wait: v modelovacím grafu je cyklus



## • Klasické synchronizační problémy

### 1) Producer - consumer

- selod s omezenou kapacitou, výrobci sbírají a konsumenți
- co když přijede výrobce a konzument zároveň? → musí se udělat mutex
- co když se selod zaplní? → blokem výrobce - čeká na konzumenta
- co když je selod prázdný? → blokem konzumenta - čeká na výrobu

### 2) večerák filozofové

→ N filozofů v kruhu - mají po pravé ruce miskičku a před sebou kolík

→ filozofové budí siemyslejí nebo jí

⇒ jak se udatat aby každý někdy jedl

(↳ co když dostanou kolod ve stejný moment?)

- pořadní pravon a čekaj na levou ⇒ deadlock

- když všechno nemá, se vrátí doleva ⇒ starvation

↳ pojde bříza, myson zablácení, ale menají se

