

CPU

- von Neumann, Harvard and Real PC architecture

- CPU architektura

ISA (Instruction Set Architecture)

↳ abstraktní specifikace mezi programátory a designery hardware

↳ registers, memory controllers, I/O

HW architektura

↳ ta implementace CPU

instrukce ----

Application Binary Interface (ABI)

↳ jak má být CPU používáno (pro designery kompilery)

↳ nepatří k ISA

↳ definice:

↳ jak používat registry

↳ implementace stacku

↳ konvence volání funkcí

Priznaky (Flags)

↳ jednobitové registry, k řízení toku a logicko aritmetických operací

↳ Z (zero)

↳ S (sign)

↳ C (carry)

↳ O (overflow) - overflow (přetečení) u čísel se znaménkem

CISC - komplexní instrukce

RISC - pamětové a aritmetické instrukce

VLIW } další

EPIC }

Orthogonalita → vlastnost ISA, říká do jaké míry lze kombinovat s jiným typem adresování a libovolným registrem

Hardwarová architektura

Hyperthreading → fyzické jádro má určitý počet výpočetních jednotek

→ pokud jádro nějakou čeká (např. na paměť), pak lze optimalizovat tím, že použije stejné výpočetní jednotky pro více vláken současně

přijímací a výstupní registry

Pipeline a superskalárita

ale zpomalí jednu instrukci

pipeline → zvýšení propustnosti, řetězení instrukcí

→ celkově bude jedna instrukce trvat takřka stejně, přestože samotná trvá déle

superskalárita → víc instrukcí zároveň během jednoho cyklu podle toho, co je to typ instrukce

Memory (paměť)

- spousta věcí vynecháno, protože probíráno v Principech počítačů

- paměť 32-bit / 64-bit (nejčastější) adresuje $2^N - 1$ adres,

1 adresa = 1 byte

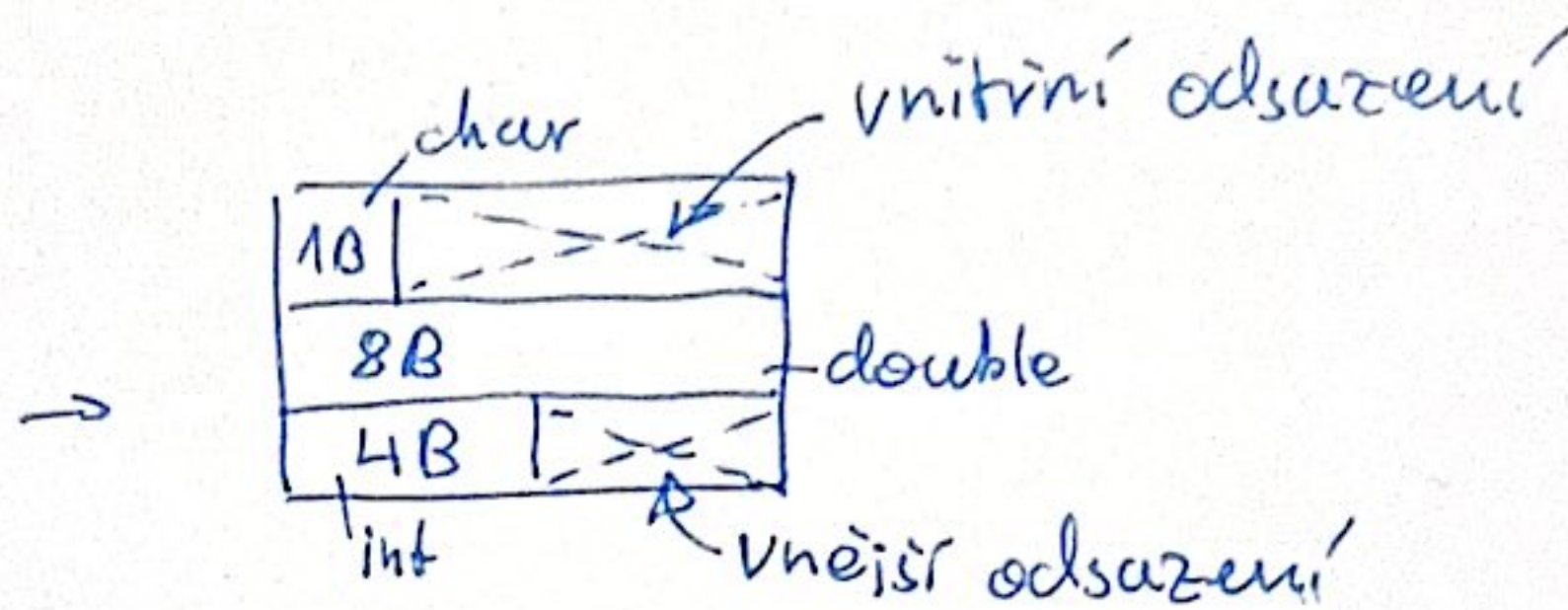
struct a {

char b;

double c;

int d;

}



↳ velikost struktury je násobek největší hodnoty

→ globální (alokováno na začátku)

→ lokální (ve stacku alokováno - pro každé volání zůstává)

→ dynamicky alokované objekty (v heapě)

Fragmentace

vnitřní → alokováno více paměti než je potřeba

vnější → vlastně mezery po blocích uvolněné paměti

Alokační algoritmy

→
dět velké
bloky, ale rychle

1. First Fit → od začátku, první blok, kam lze blok vložit

2. Next Fit → první blok od místa poslední alokace

→
několik fragmentací
na začátku

3. Best Fit → najde nejmenší dostatečně velké místo

→
pomale

4. Worst Fit → od začátku najde největší místo

Buddy Memory Allocation

- ↳ bloky velikosti 2^N ↔ adresy jsou na 2^N
- ↳ najít nejmenší 2^N blok, co stačí na alokaci, větší bloky jsou rozdělit

Cache

- ↳ řeší latenci paměti RAM
- ↳ v CPU
- ↳ časová lokalita → pokud v minulosti přistoupil, pravděpodobně přistoupí znovu
- ↳ prostorová lokalita → chce číst další data po těch co přečetl většinou

Cache line / entry

- ↳ jsou bloky, do kterých se přenáší blok z cache ca 64B

Cache Hit

- ↳ úspěšnost, že data jsou v cache, 97% (source: trust me bro)

Cache miss

- ↳ data nejsou v cache → procesor se zresetuje a načte z pomalejší paměti RAM celou cache line

Hierarchy

1. L1 → rozdělená na instrukční (L1I) a datovou (L1D)
→ uvnitř CPU, rychlá, malá
2. L2 → společná data i instrukce
→ pořadí exkluzivní pro jádro, pomalejší a větší
3. L3/LLC (last-level-cache) → nejpomalejší z cache pamětí, ale velká kapacita
→ sdílená celým procesorem → symetrická data mezi jádry

Asociativní paměť vs. RAM

↳ normální RAM = poziční adresování (fyzická adresa, moc pomalá)

Asociativní paměť (AM)

hardwarově →

↳ vyhledává podle obsahu klíče (paralelně se uvažuje a najde klíč)

↳ plně asociativní

↳ nejvyšší hit rate, těžké implementovat u větších kapacit

↳ n-cestně asociativní

↳ rozděleno do sácl

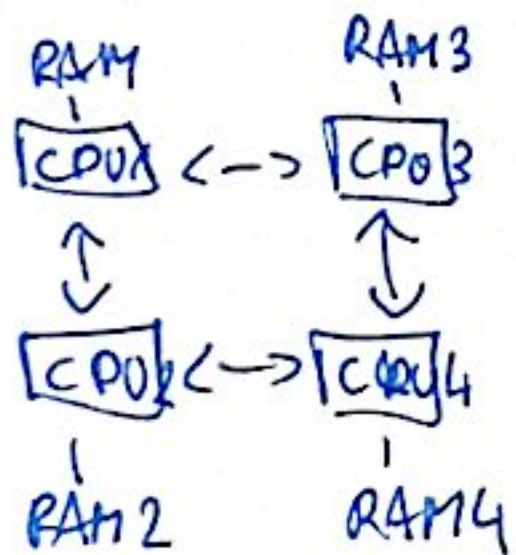
non-uniform
memory
access

→ NUMA vs. SMP

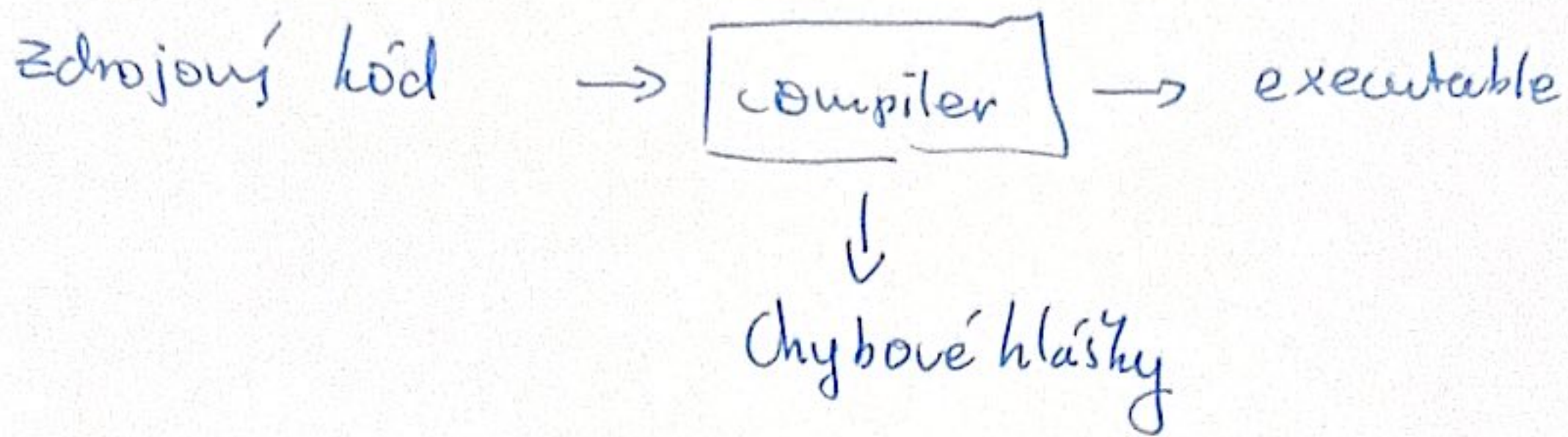
↔ symmetric multiprocessing

SMP → shodná přístupová doba k RAM

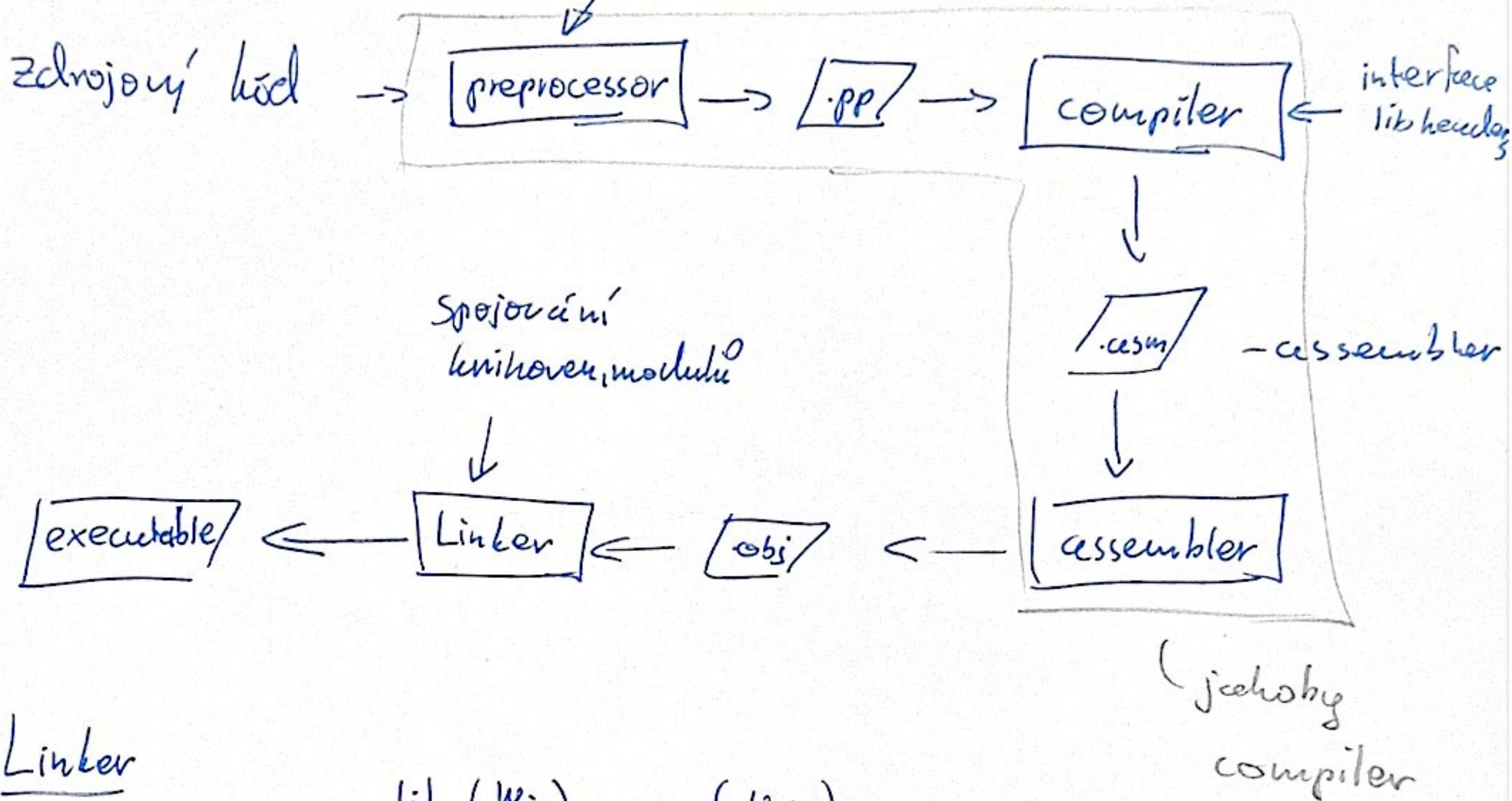
NUMA → pomalejší k RAM, ale číst od ostatních CPU rychlejší



Progra mowai' jazyky



- prostě executable je jednoznačný
expand maker, prostě substituce



Linker

compiler

lib (Win) .a (Unix)

statické → zahrnuje kód a vloží přímo do výsledného executable, bude fungovat, ale větší

dynamické → vloží pouze odkazy, závislosti na správné knihovně

dll (Win) .so (Unix)

Run-time

statická podpora → rozhraní knihoven, zajištěna v době překladu

dynamická podpora → organizace, paneti, housebuktory, ...

aktivacní zážnam a celuje se pomocí frame pointem

↳ kontext potřebný pro both funkce - návratová hodnota,
ukládání stavu stroje, kontrolní odstup, lokální proměnné
↳ neaktivní záznam

rolaci' homenne

4) jak si funkce vypočítá: dle

public name mangling

↳ upravování názvů globálních funkcí pro rozlišitelnost

Caller (připraví ...) → Callee (frame pointer ...) → Návrat (číslo ...)

call by value - funkce s hodnotou
call by reference - funkce s pointerem } v parametru

static data , stack , heap
↳ allocate , deallocate

garbage collection

↳ for heap

↳ strategies - tracing (náročná)
- reference counting (problém s cykly)

source code portability

binary portability

speed problems solve : just-in-time JIT
ahead-of-time AOT

Operační systém

role

- ↳ abstraktní stroj → poskytuje kernel API nebo-li syscalls
- ↳ správa prostředků → dělí fyzický HW mezi více procesů

Režimy procesoru

1. Uživatelský mód (ring 3)

- ↳ standardní aplikační kód, nepokouší přístup k systémovým registrům a určitým instrukcím

2. Kernel Mode (ring 0)

- ↳ neomezený přístup

přepnutí → syscall a sysret

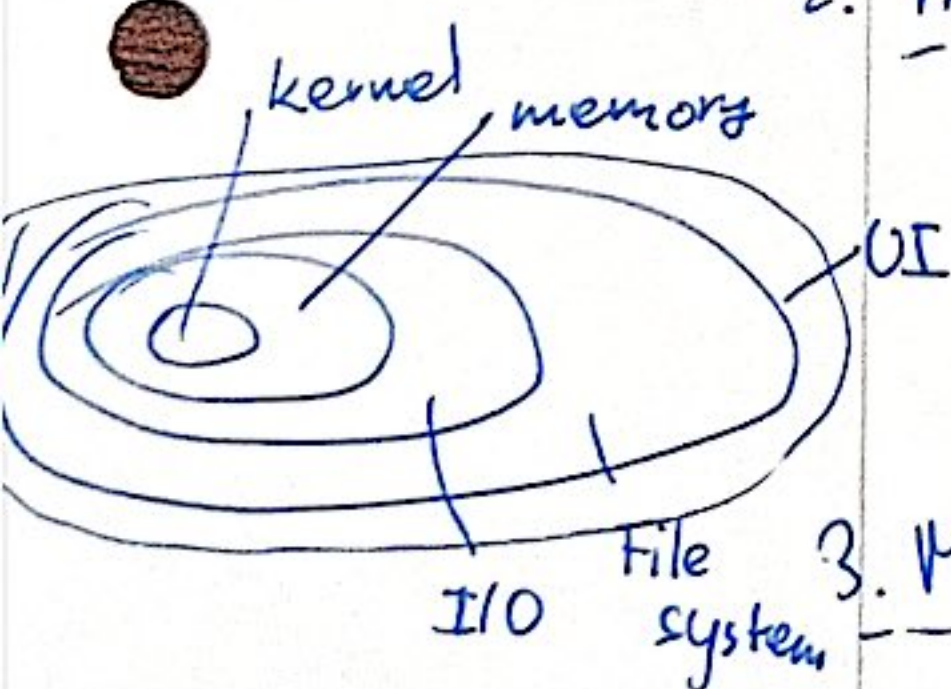
1. Monolitická architektura

- ↳ vše běží propojené v ring 0
- ↳ rychlé, ale hodně chyb

kdysi Linux →

2. Vrstvená architektura

- ↳ vrstva $n+1$ může používat jen služby, co poskytuje vrstva n



3. Mikrojádno

- ↳ ring 0 je minimalistické
- ↳ vše ostatní běží jako izolované služby v user mode
- ↳ méně rychlé, ale bezpečné

Zařízení

SW/OS

enumerate devices on boot

- ↳ device controller, device driver, bios/uefi

- ↳ topologie: bus, ring, star, tree

- ↳ jak Device I/O → speciální instrukce
→ memory-mapped I/O (devices)

Zařízení - komunikace

↳ polling - CPU checkuje

↳ interrupt - zařízení zavolá CPU a ten přeruší execution, CPU má na to alespoň 1 pin

↳ DMA (direct memory access) → bez CPU

Typy přerušení

1. externí → HW zavolá CPU na IRQ (interrupt request)

2. (hardware) výjimka → nečekaně přes instrukci

3. software → pomocí speciální instrukce (např. int)

Interrupt Request (IRQ) Handling

1. CPU získá adresu přerušení

2. CPU přeruší proud instrukcí

3. CPU se přepne do kernel módu a spustí obsluhu přerušení

4. OS aktualizuje stav a obsluhuje zařízení

5. OS obnoví stav CPU

6. CPU obnoví kritický stav

Processing

program → instrukce

process → instance programu

thread → jedna aktivita, posloupnost instrukcí, tedy uložení ~ kódu

Parent-child process relation

Linux → child has its parent id, terminated process is zombie, after

killing process, children are orphaned

Windows → není vztah žádný

SIGCHLD

↳ child notifies
of its death

Processing

scheduler - plánuje procesy

multitasking - víc procesů najechnou

multiprocessing - více jader v procesoru, těžší pro scheduler

context-switch → měnit scheduling unit, náročné

Units of scheduling

states

created, ready, running, blocked, terminated

proces si
určuje

→ kooperativní → dohodnou se, kdo kdy může přistupovat k CPU

preemptivní → všichni mají maximální dovolený čas, OS určí

priority k plánování

↳ statická = při vytvoření procesu, nemění se

↳ dynamická = zvyšuje délku čekáním

Plánovací algoritmy

nepreemptivní →

1. FCFS (first come, first serve)

2. SJF (shortest job first)

3. LJF (longest job first) - zřídka

preemptivní →

1. Round Robin → přestě fronta a max. čas na CPU

2. Víceúrovňová fronta → více front a čas se mění podle toho, kolik kdo potřebuje, posouvá nahoru x dolů

3. CFS (completely fair scheduler)

↳ v Linuxu do 2024, černé-černé stromy

Meziprocesová komunikace

↳ pipes / sockets - (TCP over loopback)

↳ sdílená paměť

↳ signály

Soubor

- ↳ logická jednotka abstrahující fyzické uložení dat
- ↳ file handle, buffering
 - ↳ sequential / random access

↳ memory mapping, async file I/O

directory

↳ prostě složka

file storage → tradiční úložiště (FS ~ RAM / HDD)

→ network storage

→ virtual system files → $\left. \begin{array}{l} /dev/null \\ /dev/urandom \end{array} \right\} \dots$

file links

↳ víc adresářů ukazuje na stejný fyzický soubor = hard link

↳ soubor, obsah je cesta k jinému souboru = symlink

Architektura souborových systémů (FS)

FAT → má na začátku tabulku (file allocation table) — nepoužívá

Ext2 → second extended file system

↳ rozděleno do blokových skupin, každá skupina má
zálohu superbloku, bitmapu volných bloků a inodů, tabulku inodů a
data.

HDD

↳ seek time, rotation latency, transfer time

SSD → rychlejší se nicí u různých FS podle přepisování
→ flash, grid

Plánování diskových operací

↳ FCFS, SSTF, SCAN, C-SCAN, LOOK/C-LOOK

RAID → rozdělení disků na mirror RAID 1, RAID 0 (interleaving), RAID 5/6

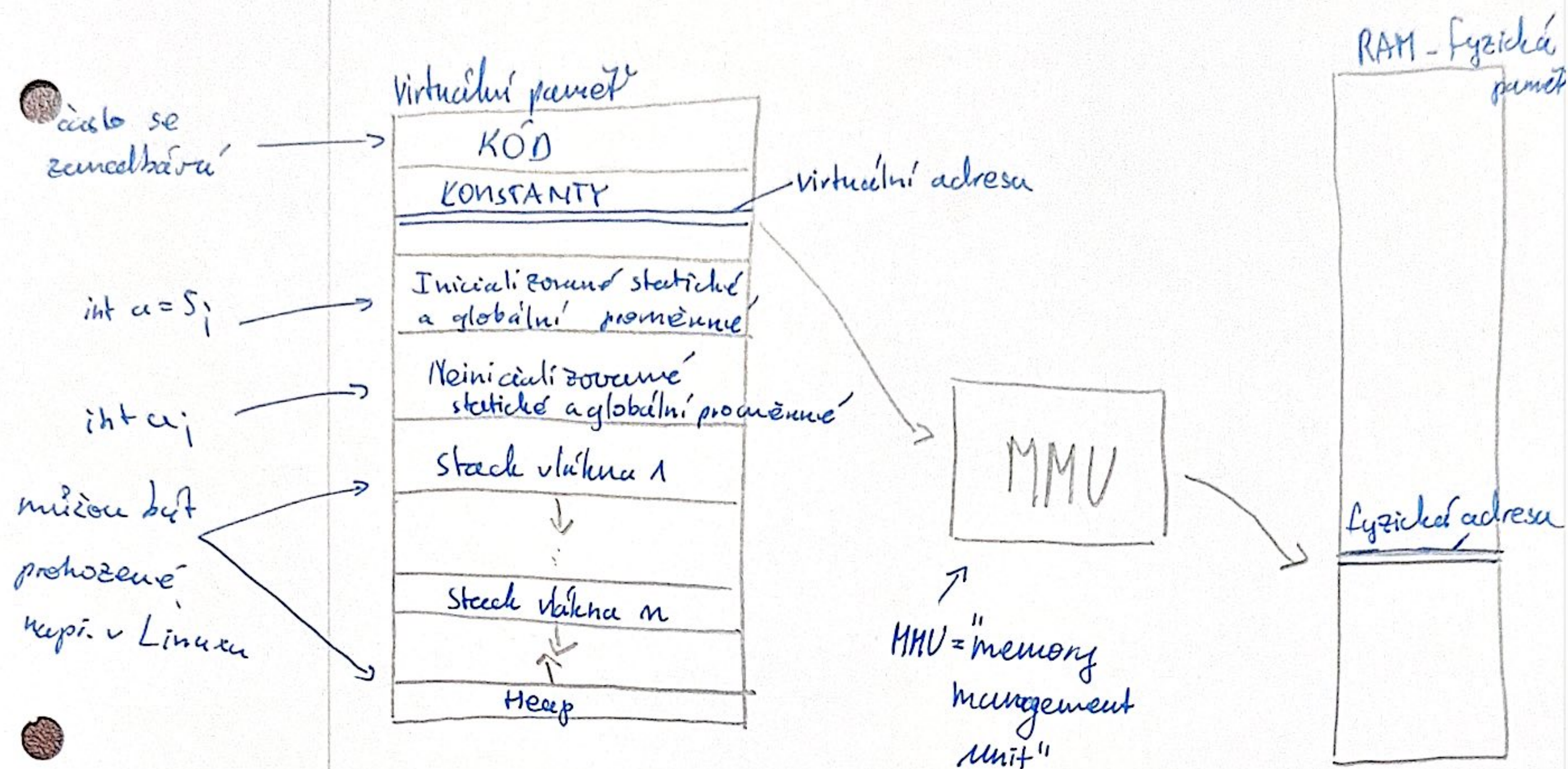
↳ velké
bezpečí

↳ pro bezpečí

Partitions → dělení jednoho disku, každý oddíl i vlastní FS

Nes se používá
Ext3

Virtuální paměť



Stack (Záložník)

↳ ukládá se sem proměnné ve funkcích lokální a dočasné a návratové adresy funkcí

Heap (haldce)

↳ ukládá se sem dynamicky alokované objekty třeba objekty, pole ...

↳ v C/C++ musí být manuálně dealokována pomocí `free()`, `delete`, ...

MMU

↳ převádí virtuální adresu na fyzickou

↳ 2 přístupy: stránkování a segmentace

Důvod proč

↳ Virtuální paměť může být větší než fyzická (chuze), připojí se například další paměť / úložiště = swap pro backup

program indexuje od 0 a myslí si, že má k dispozici celou paměť, ale OS alokuje fyzické rámce postupně

Segmentace

↳ rozdělit paměť podle významu dat na celky, které se identifikují číslem segmentu a offsetem

↳ v rámci virtuální adresy

↳ pak pomocí tabulky segmentů → obsahuje fyzickou bázeovou adresu
→ obsahuje segmenty, limity a offset

↳ MMU → zkontroluje limity a $(\text{fyzická adresa}) = (\text{bázeová fyzická adresa}) + (\text{offset})$

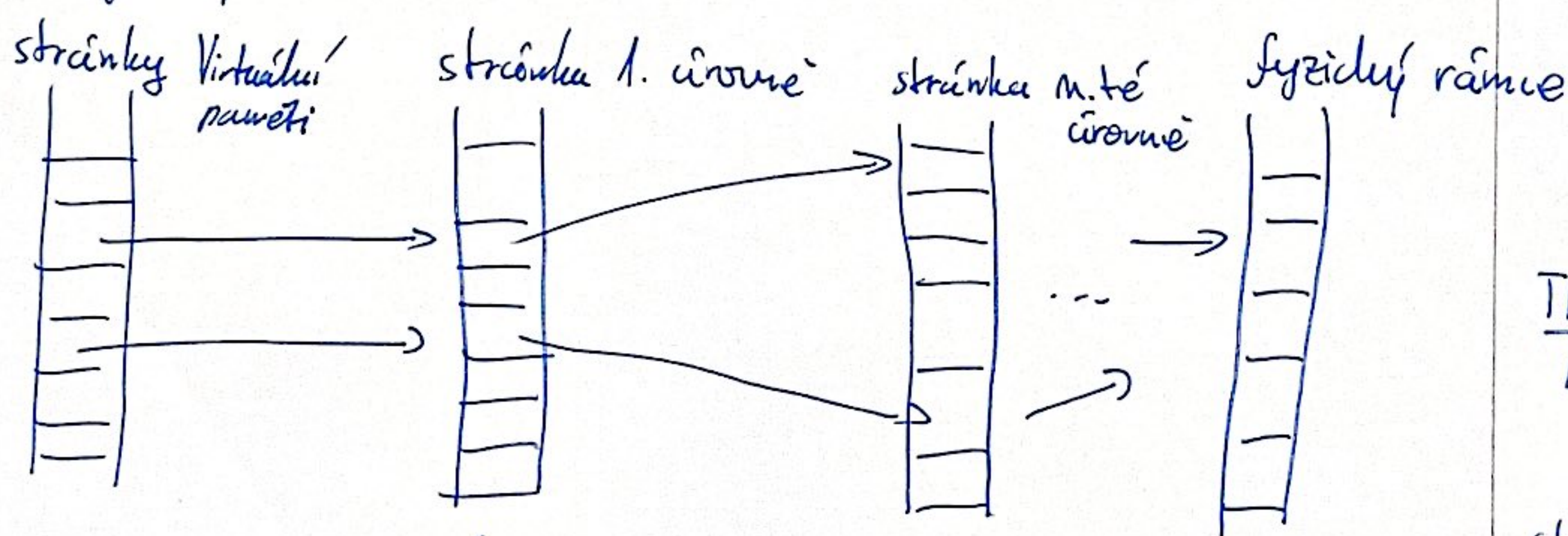
Stránkování

↳ virtuální paměť je rozdělena na stránky (velikosti 2^n)

a fyzická paměť na rámce (ekvivalent 1:1 $\Rightarrow$ 1 rámec = 1 stránka)

↳ stránkovací tabulka (1 až více úrovní)

↳ obsahují adresu na další stránku a poslední úroveň na fyzický rámec

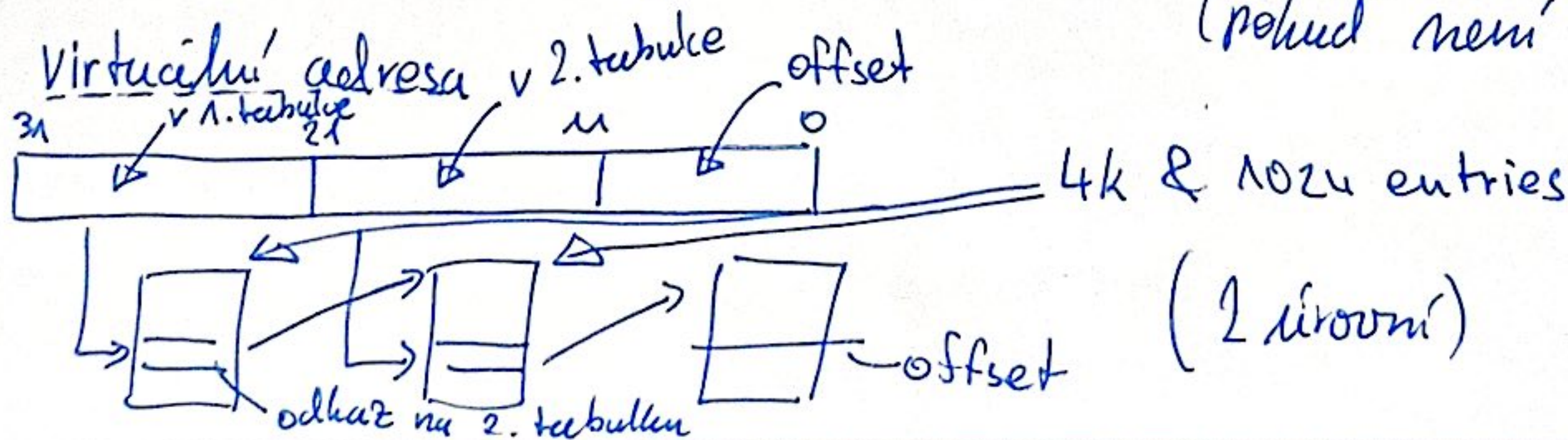


↑
pořadí v paměti

↳ nemusí být v paměti (pokud není = page fault)
↳ rychlejší

TLB

↳ buffer na ukládání adres stránek na rámce bez nutnosti překládání



Stránkování - přehled adres

- zkontrolovat TLB (Translation Lookaside Buffer)
- update A(cessed) and D(irty) bits in page table / TLB
 - ↳ $A=1$ (nastaveno) pokud se stránka nedávno přistoupila, časem se nuluje
 - ↳ $D=1$ (nastaveno) pokud změněna v TLB, při načítání je třeba upravit i na disku (u $D=0$ ne)
- pak prostě přehled těch stránek vložit do TLB

Při chybě (Page Fault exception)

Pokud • nepovolen vstup → zamítnutí

- validní adresa, ale stránka není namapována
1. najít volný rámec nebo nějaký nahradit
- vytvořit stránku další úrovně

2. opakovat instrukci

Algoritmy na nahrazení stránek

1. Clock → projde stránky

→ pokud $A \neq 0$ → nastav $A=0$

• $A == 0$ → vyber a nahraď

not recently used

2. NRU → periodicky nastavuje $A=0$

→ pomocí A a D vytvoří 4 třídy

(někdy)

→ odsud vezme a nahradí

	A	D
0	0	0
1	0	1
2	1	0
3	1	1

least recently used

3. LRU → nahradí stránku, co nebyla použita nejdříve, nějak pomocí SW nebo HW

not frequently used

4. NFU → aproximace LRU

→ počítadlo, inkrementuje periodicky u $A == 1$, ale hned se nastaví na $A=0$

→ nahradí se s nejmenším počítadlem

existuje optimální, ale záleží na programu a t.d. →

Pokročilejší stránkování

Sdílená paměť (shared memory)

↳ více procesů sdílet stejná data ve fyzické paměti, ze stejného rámce (stránky odkazují na stejné místo)

Soubory mapované do paměti

↳ soubory k dispozici ve virtuální paměti (podobný princip co předchozí)

Virtuální stroj a kontejnery

1. Plná virtualizace

↳ abstrakce na úrovni hardwaru, hypervisor emuluje hardware

2. Procesní virtuální stroj

↳ abstraktní vrstva nad běžným OS, nezávislost zkompileovaného

kódu → Java VM, .NET CLR

↳ instrukce ze běhu → JIT (just-in-time)

→ AOT (ahead-of-time)

3. Container

↳ abstrakce nad OS, izolované instance uživatelského prostoru → málo náročné

Paralelní programování

parallelismus

↳ provádění výpočtu v tomtéž čase (na různých vláknech)

konkurentní

↳ multitasking na jednom CPU, rychlý kontext switch, iluze

Race condition

↳ více vláken současně přistupuje ke stejné paměti a alespoň jeden z nich ji modifikuje

↳ plánovač OS → řeší, aby se nic nepokazilo

kritická sekce → část kódu, kde může dojít k race condition

vzájemné vyloučení → max. 1 proces (jednotek plánování) může přistupovat současně

Synchronizace procesů → dohoda, kdo přistupuje kdy

Synchronizace dat → pomocí synchronizace procesů, aby data byla korektní

Synchronizující primitiva

Aktivní (např. Spin-Lock pomocí CAS (compare and swap) = TSL (test and set))

```
bool cas (T* var, T old, T newVal):  
{  
    if (*var != old) return false;  
    *var = newVal;  
    return true;  
}
```

neustále se ptá

HW
implementování

atomické
instrukce

Poslání (např. semafor) - uspi se a pak se vzbudí

```
void down() {  
    if (counter > 0) --counter;  
    else {  
        queue.push(my VS);  
        my VS.block();  
    }  
}
```

užití

```
semafor.down();  
#kritická sekce  
semafor.up();
```

```
void up() {  
    if (counter == 0 && !queue.isEmpty()) {  
        VS = queue.pop();  
        VS.unblock();  
    }  
    else ++counter;  
}
```

Mutex → semafor s čítačem $\wedge$ dostupné jen jednomu vláknu

Bariéra → synchronizační bod

→ jakmile sem dorazí vlákno, tak nemůže pokračovat dále i ostatních N vláken neeloraž

Deadlock

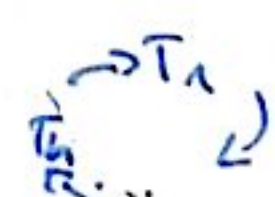
↳ když se ječho nebo více vláken trvale zablokuje

Podmínky, kdy k němu dojde (Coffmanovy podmínky) - všechny 4

1. Mutex → část kódu může je 1 vlákno

2. Hold and wait → vlákno má 1 zdroj a čeká na přiřazení dalšího zdroje

3. No preemption → OS nemá způsob jak vláknům zdroj vzít

4. Circular wait → vlákno T_1 čeká na T_2 , T_2 na T_3 , ..., až T_n na T_1 

Ukázky: ^aProducer-consumer, ^bDining philosophers, ^cReaders and writers, ^dSleeping barber