

# B4B35OSY: Operační systémy

## Lekce 1. Úvod do operačních systémů

Petr Štěpán  
[stepan@fel.cvut.cz](mailto:stepan@fel.cvut.cz)



24. září, 2020

# Outline

## 1 Úvod

- Cíle předmětu

## 2 Malý návod na použití školy

## 3 Co je operační systém

## 4 OS (nejen osobního) počítače

# Obsah

## 1 Úvod

- Cíle předmětu

## 2 Malý návod na použití školy

## 3 Co je operační systém

## 4 OS (nejen osobního) počítače

# B4B35OSY – Operační systémy

Přednášející:

**Michal Sojka**, CIIRC

`Michal.Sojka@cvut.cz`

**Petr Štěpán**, FEL, katedra kybernetiky

`Stepan@fel.cvut.cz`

# Materiály

- Silberschatz A., Galvin P. B., Gagne G.: *Operating System Concepts*  
<http://codex.cs.yale.edu/avi/os-book/OS7/os7c/index.html>
- Tanenbaum, Andrew S a Albert S Woodhull: *Operating systems design and implementation*. 3rd ed. Upper Saddle River: Prentice-Hall, c2006, xvii, 1054 s. ISBN 0131429388
- <http://wiki.osdev.org/>
- <http://hypervisor.org/>
- YouTube lectures (anglicky):
  - CS 162 – UC Berkeley
  - OS-SP06 – Surendar Chandra – UC Berkeley
  - MIT 6.004

# Organizace předmětu

- Souhrnná podrobná literatura v češtině není
- Tyto prezentace (stránka předmětu  
<https://cw.fel.cvut.cz/wiki/courses/b4b35osy>)
- Cvičení částečně seminární, více samostatná práce, nutná domácí příprava
- Hodnocení:
  - Body ze cvičení
    - Úlohy celkem až 50 bodů
    - Aktivita při hodině max 10 bodů
  - Písemná zkouška max 30 bodů
  - Ústní část max 10 bodů – dobrovolná (nutná pro A)

# Obsah

## 1 Úvod

- Cíle předmětu

## 2 Malý návod na použití školy

## 3 Co je operační systém

## 4 OS (nejen osobního) počítače

## Cíle předmětu

Podle Hospodářských novin se Informatika vyučuje nejlépe na FEL, ČVUT  
(22. 1. 2015)

- OS patří k základům informatiky
- Poznat úkoly OS a principy práce OS
- Využívat OS efektivně a bezpečně

Co NENÍ cílem tohoto předmětu

- Naučit Vás jak napsat aplikaci pod (X)Windows
- Naučit triky pro konkrétní OS
- Vytvořit OS – na to je málo času

# Proč studovat OS

- Pravděpodobně nikdo z vás nebude psát celý nový OS
- Proč tedy OS studovat?
  - Každý ho používá a jen málokdo ví jak pracuje
  - Jde o nejrozsáhlejší a nejsložitější IT systémy
  - Uplatňují se v nich mnohé různorodé oblasti
    - softwarové inženýrství,
    - netradiční struktury dat,
    - sítě, algoritmy, ...
- Čas od času je potřeba OS upravit
  - pak je potřeba operačním systémům rozumět
  - psaní ovladačů, ...
  - Mnoho programátorských problémů lze na nižší úrovni vyřešit snadněji a efektivněji
- Techniky užívané v OS lze uplatnit i v jiných oblastech
  - neobvyklé struktury dat, krizové rozhodování, problémy souběžnosti, správa zdrojů, ...
  - mnohdy aplikace technik z jiných disciplín (např. operační výzkum)
  - naopak techniky vyvinuté pro OS se uplatňují v jiných oblastech (např. při plánování aktivit v průmyslu)

# Naučit se lépe programovat

- Programování má různé podoby/úrovně (měli byste se seznámit se všemi):
  - Integrace high-level knihoven (mnohé webové a mobilní aplikace)
  - Aplikační programování (AI, počítačové hry, ...) – obsahují vlastní algoritmy
  - Nízko-úrovňové programování (OS, embedded systémy, ...) – pomezí SW a HW
- *"You might not think that programmers are artists, but programming is an extremely creative profession. It's logic-based creativity."*

—John Romero

# Naučit se přehledně programovat

- *"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."*

—Martin Fowler

- V jednoduchosti je krása
  - řešení úlohy továrna z loňského roku – na 200 ale i 2000 řádek kódu
- V dnešní době je program způsob záznamu informací/znalostí
  - V jedné pekárně se porouchal stroj a museli povolat pekaře-důchodce, protože nikdo jiný neznal recept na chleba ani nerozuměl programu stroje.
- Nepište kód pro počítač, ale pro lidi, aby byl srozumitelný a znalosti tam byly na první pohled viditelné

- *"Programming is the art of algorithm design and the craft of debugging errant code."*

—Ellen Ullman

# Naučit se komentovat program

- Používejte vhodně pojmenované proměnné a funkce
- Komentujte co má funkce dělat a naznačte jak to dělá
- Komentujte jen to, co nelze vyjádřit programovacím jazykem
  - $i = 1; //$  do proměnné i přiřadíme hodnotu 1 – NE!
- *"The cleaner and nicer the program, the faster it's going to run. And if it doesn't, it'll be easy to make it fast."*

—Joshua Bloch
- Opět – Nepište kód pro počítač, ale pro lidi, aby byl srozumitelný a znalosti tam byly na první pohled viditelné

# Obsah

## 1 Úvod

- Cíle předmětu

## 2 Malý návod na použití školy

## 3 Co je operační systém

## 4 OS (nejen osobního) počítače

# Covid-19

- Výjimečný stav požaduje výjimečné výkony
- Větší nároky:
  - soustředění
  - iniciativu
  - domácí práci
  - organizace práce
- Řešení:
  - zvýšená komunikace, jak mezi Vámi tak směrem k nám
  - orientovat se v již probíhajících debatách

# Cíle vzdělávání

- V obecné rovině
  - Naučit kriticky myslit
  - Naučit hledat zákonitosti
- V konkrétní rovině
  - Předat nějaké konkrétní znalosti (co je posix, cache, sběrnice)
  - Předat nějaké konkrétní dovednosti (jak se programuje, jak efektivně vést projekt)

# O dobrém a špatném učení

## ■ Povrchní přístup k učení

- Úkoly dělám, abych splnil jejich zadání a dostal body
- Výsledkem je zpravidla memorování

## ■ Hloubkový přístup k učení

- Úkoly dělám, abych splnil jejich účel
- Výsledkem je zpravidla porozumění
- Navíc je nutné najít účel úloh

# Proč porozumět a ne memorovat

- Schopnost spojit nové a dřívější znalosti
  - Pomáhá v chápání nových znalostí
  - Pomáhá odstranit chybné znalosti
- Schopnost použít znalosti
  - Znalosti lze spojit s každodenní zkušeností
- Schopnost uchovat znalosti
  - Dobře spojené a pochopené znalosti se pamatují déle
- Volba je na Vás !

# Co bude na přednáškách

- Výkladu se nedá uniknout
  - Náplň je většinou předem k dispozici
- Je to jako kino, ne? Návštěva kina je:
  - Pasivní zážitek s občas zajímavým příběhem
  - Nemusíte příliš přemýšlet
  - Desítky miliónů \$ vynaložené na udržení Vaší pozornosti
- Aktivní učení
  - Charles Lin: Active Learning in the Classroom
  - <http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Learn/active.html>

# Co bude na přednáškách

- Když chodíte na přednášky
  - očekává se, že se něco naučíte
- V čem je problém
  - Látka je složitá, ale při poslouchání to člověku nepřijde
  - Většina věcí se jeví logická – myšlenkové zkratky
  - Pro zvládnutí je nutné se jí nějakou dobu věnovat i po přednášce
    - ACM/IEEE CS Curriculum: na 1 hodinu přednášky v bakalářském studiu připadají 2–3 hodiny domácí přípravy

# Jak se něco na přednášce naučit

- Neusnout
  - Bez ohledu na to, jak těžké to může být
  - Kdo spí, ten se nic nenaučí a přichází o souvislosti
- Chodit pravidelně
  - Nová látka staví na předchozích základech
  - Naučíte se lépe rozumět přednášejícímu
  - Když jsem minule nebyl, alespoň si přečíst přednášky
- Aktivně poslouchat
  - Nejlépe se nové věci naučíte při hledání vlastního vysvětlení, jak věci fungují
  - Dává smysl to co slyšíte?
  - Byli byste schopni to vysvětlit někomu, kdo na přednášce nebyl?
- Pokud něco nedává smysl
  - Zapište si co Vám nedává smysl
  - Zkuste vymyslet otázku, jejíž zodpovědění by věci vyjasnilo a položte ji přednášejícímu

# Kdy a jak se ptát

- Když Vaše představa neodpovídá tomu co slyšíte
  - Nebo když Vám chybí část „skládanky“
  - Na konci přednášky byste měli být schopni položit několik otázek, alespoň upřesňujících
    - „Myslím si, že říkáte ..(vlastními slovy)..., je to tak?“
- Než se zeptáte, zkuste si odpovědět
  - Pokud nevíte, nebo si nejste jisti, zeptejte se
- Při hledání otázek začnete pozorněji poslouchat
  - Začnete poslouchat s cílem se něco naučit
  - Naučíte se klást užitečné dotazy

# Obsah

## 1 Úvod

- Cíle předmětu

## 2 Malý návod na použití školy

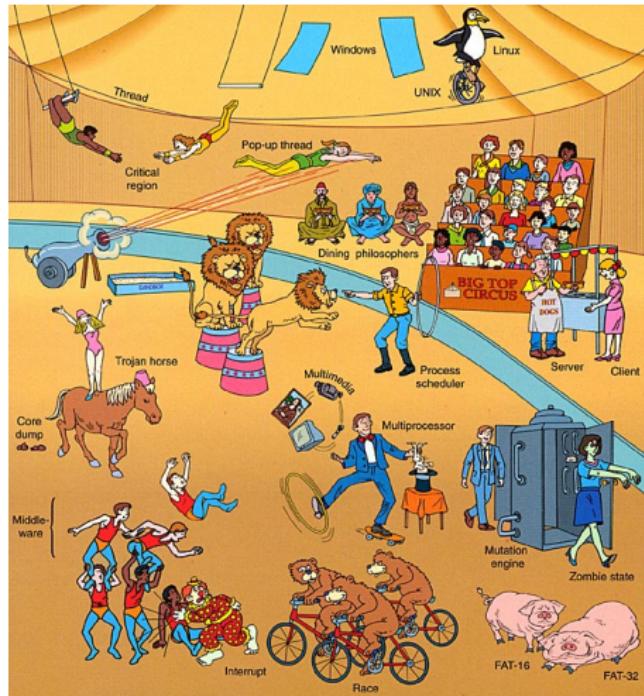
## 3 Co je operační systém

## 4 OS (nejen osobního) počítače

# Co je operační systém

## Úkoly OS:

- Spouštět a dohlížet uživatelské programy
- Efektivní využití HW
- Usnadnit řešení uživatelských problémů
- Učinit počítač (snáze) použitelný
- Umíte použít počítač bez OS?



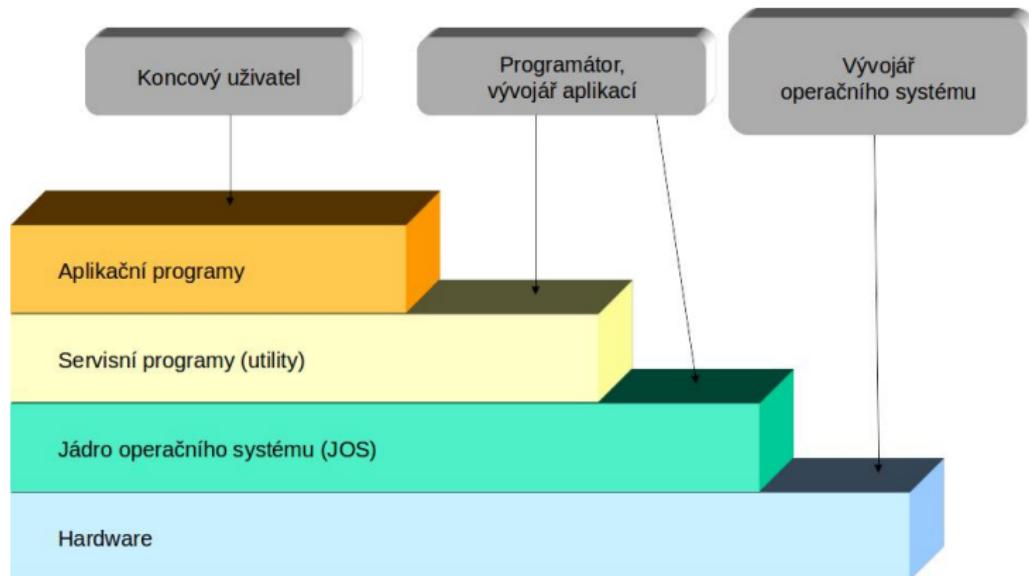
# Co je operační systém

- Neexistuje žádná obecně platná definice
- Několik koncepcí pojmu OS
  - systémové (jen jádro a s ním související nadstavby)
  - „obchodní“ (to, co si koupíme pod označením OS)
  - organizační (včetně pravidel pro hladký chod systému)
- OS jako rozšíření počítače
  - Zakrývá komplikované detaily hardware
  - Poskytuje uživateli „virtuální stroj“, který se snáze ovládá a programuje
- OS jako správce systémových prostředků
  - Každý program dostává prostředky v čase
  - Každý program dostává potřebný prostor na potřebných prostředcích
  - Prostředky jsou CPU, paměť, periférie

# Co je operační systém

V této přednášce budeme brát operační systém jako jádro operačního systému

- ostatní (tzv. systémové) programy lze chápat jako nadstavbu jádra
- GUI – Windows je grafická nadstavba systémových programů

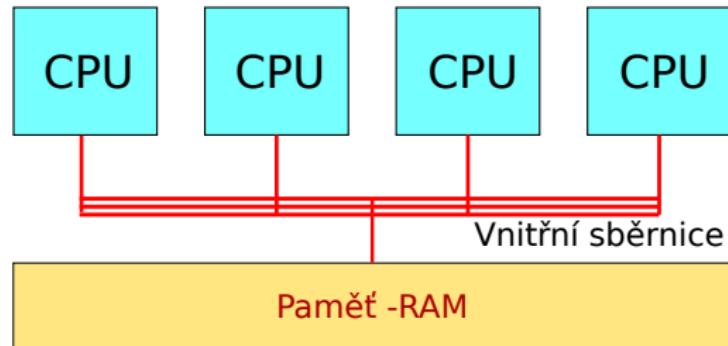


## Různorodost OS

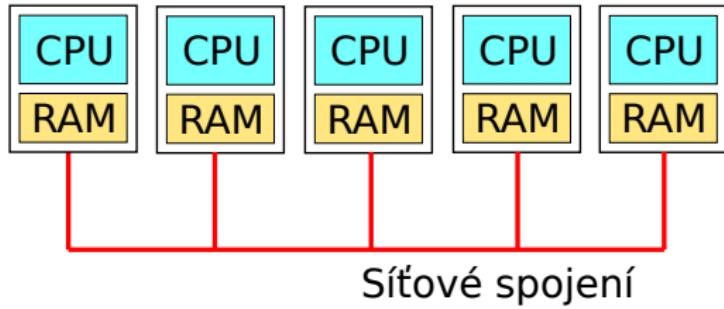
- OS „střediskových“ (mainframe) počítačů – dnes již historický pojem
- OS superpočítačů (5 mil. jader, 200 PFlops, 13 MW příkon)
- OS datových a síťových serverů
- OS osobních počítačů a pracovních stanic
- OS reálného času (Real-time OS – řízení letadel, vlaků, raket, družic, apod.)
- OS přenosných zařízení – telefony, tablety
- Vestavěné OS (tiskárna, pračka, telefon, ...)
- OS čipových karet (smart card OS)
- ... a mnoho dalších specializovaných systémů

# Paralelní a distribuované systémy

Těsně vázaný  
multiprocesorový  
systém



Distribuovaný systém  
typu klient-server



# Systémy reálného času – RT

- Nejčastěji řídicí zařízení v dedikovaných (vestavěných) aplikacích:
  - vědecký přístroj, diagnostický zobrazovací systém, systém řízení průmyslového procesu, monitorovací systémy
  - obvykle dobře definované pevné časové limity
  - někdy také subsystém univerzálního OS
- Klasifikace:
  - striktní RT systémy – Hard real-time systems
    - omezená nebo žádná vnější paměť, data se pamatují krátkodobě v RAM paměti
    - protipól univerzálních OS nepodporují striktní RT systémy
    - plánování musí respektovat požadavek ukončení kritického úkolu v rámci požadovaného časového intervalu
  - tolerantní RT systémy – Soft real-time systems
    - použití např. v průmyslovém řízení, v robotice
    - použitelné v aplikacích požadujících dostupnost některých vlastností obecných OS (multimedia, virtual reality, video-on-demand)
    - kritické úkoly mají přednost „před méně šťastnými“

## Více úloh současně – Multitasking

- Zdánlivé spuštění více procesů současně je nejčastěji implementováno metodou sdílení času tzv. Time-Sharing Systems (TSS)
- Multitasking vznikl jako nástroj pro efektivní řešení dávkového zpracování
- TSS rozšiřuje plánovací pravidla
  - o rychlé (spravedlivé, cyklické ) přepínání mezi procesy řešícími zakázky interaktivních uživatelů
- Podpora on-line komunikace mezi uživatelem a OS
  - původně v konfiguraci počítač – terminál
  - v současnosti v síťovém prostředí
- Systém je uživatelům dostupný on-line jak pro zpřístupňování dat tak i programů

# Obsah

## 1 Úvod

- Cíle předmětu

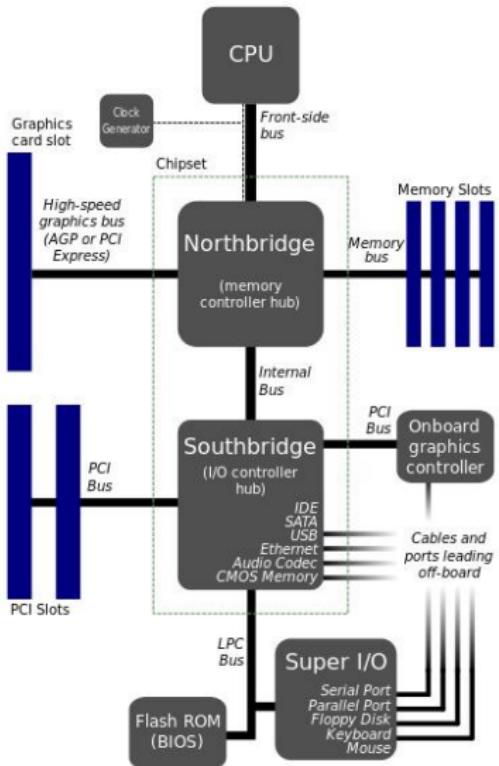
## 2 Malý návod na použití školy

## 3 Co je operační systém

## 4 OS (nejen osobního) počítače

# Osobní počítač

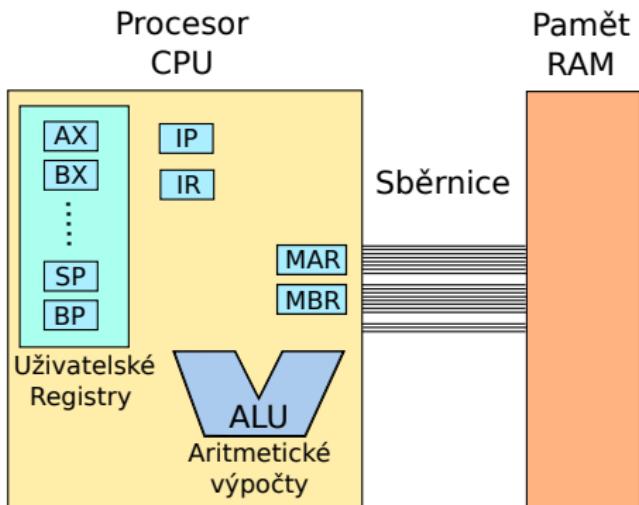
- Základem počítače je procesor – CPU
- Procesor je připojen sběrnicemi (bus, interconnect) k ostatním periferiím počítače – paměti, grafickému výstupu, disku, klávesnici, myši, síťovému rozhraní, atd.
- Činnost sběrnice řídí arbitr sběrnice



# Procesor – CPU

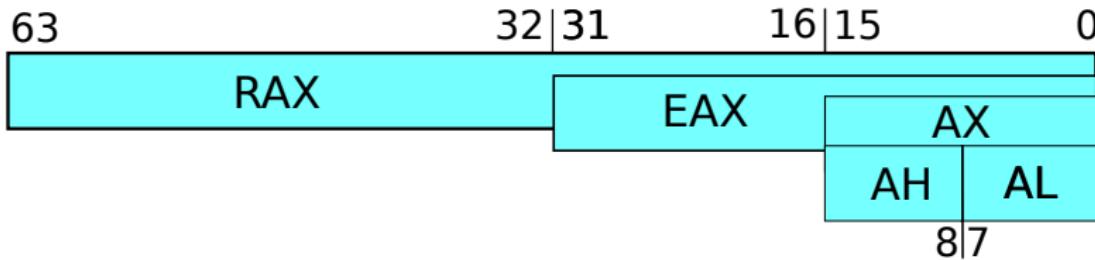
## ■ Základní vlastnosti:

- šířka datové a adresové sběrnice
- počet vnitřních registrů
- rychlosť řídicího signálu – hodiny
- instrukční sada



# Procesor – x86/AMD64

- Přehledný popis –  
[https://en.wikibooks.org/wiki/X86\\_Assembly](https://en.wikibooks.org/wiki/X86_Assembly)
- Všechny registry vzhledem ke zpětné kompatibilitě jsou 64/32/16/8 bitové

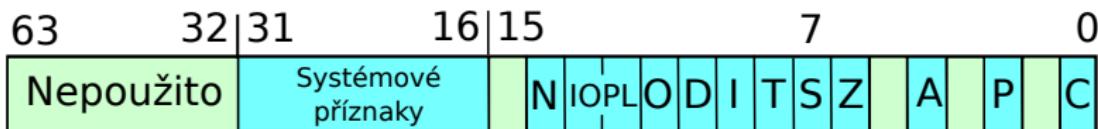


## Řídicí a stavové registry

- EIP/RIP – instruction pointer – adresa zpracovávané instrukce
- EIR/RIR – instruction registr – kód zpracovávané instrukce
- EFLAGS/RFLAGS – stav procesoru povoleno/zakázáno přerušení, system/user mód, výsledek operace – přetečení, podtečení, rovnost 0, apod.

# Registr FLAGS

## RFLAGS registr



C – Carry flag

P – Parity flag

Z – Zero flag

S – Sign flag

O – Overflow flag

I – Interrupt enable

T – Trap flag

IOPL – I/O privilege level

A – Adjust flag

# Režimy práce procesoru

## FLAGS registr

- Dva režimy práce procesoru IOPL – základ hardwarových ochran
  - CPL0<sup>1</sup> = privilegovaný (systémový) režim
    - procesor může vše, čeho je schopen
  - CPL3 = uživatelský (aplikáční) režim
    - privilegované operace jsou zakázány
- Privilegované operace
  - ovlivnění stavu celého systému (halt, reset, Interrupt Enable/Disable, modifikace Flags, modifikace registrů MMU )
  - instrukce pro vstup/výstup (in, out)
- Přechody mezi režimy
  - Po zapnutí stroje systémový režim
  - Přechod do uživatelského – modifikace Flags (popf nebo reti)
  - Přechod do systémového – pouze přerušení vč. programového

---

<sup>1</sup>Current privilege level

# Procesor – x86/AMD64

## Uživatelské registry

- programově dostupné registry pro ukládání hodnot programu *eax*, *ebx*, *ecx*, *edx*
- registry umožňující uchovat hodnotu, nebo ukazatel do paměti *esi*, *edi*, *ebp*
- *esp* – stack pointer – ukazatel zásobníku - detailněji dále
- AMD64/X86-64 přidává 8 dalších registrů *r8-r15*, ve formě *r8b* nejnižší bajt, *r8w* nejnižší slovo (16 bitů), *r8d* – nižších 32 bitů, *r8* – 64 bitový registr

# Instrukce – x86/AMD64

## Instrukce "ulož hodnotu"

(běžně se používají dvě různé syntaxe pro zápis assembleru)

### AT&T

movq zdroj 64b, cíl	mov cíl, zdroj
movl zdroj 32b, cíl	
movw zdroj 16b, cíl	
movb zdroj 8b, cíl	
registry se značí %ax	pouze ax
hodnoty \$, hex 0x	číslo, hex postfix h
movl \$0xff, %ebx	mov ebx, 0ffh

### Intel

# Instrukce – x86/AMD64

Ulož hodnotu na adresu (odkaz do paměti)

AT&T

```
movl (%ecx), %eax  
movl 3(%ebx), %eax  
movl (%ebx, %ecx, 0x2), %eax  
movl -0x20(%ebx, %ecx, 0x4), %eax
```

Intel

```
mov eax, [ecx]  
mov eax, [ebx+3]  
mov eax, [ebx+ecx*2h]  
mov eax, [ebx+ecx*4h-20h]
```

- odkaz má 4 složky: *základ+index \* velikost + posun*
- pole struktur o velikosti *velikost*, *základ* je ukazatel na první prvek, *index* říká, který prvek chceme a *posun*, kterou položku uvnitř struktury potřebujeme.
- není potřeba použít všechny 4 složky

# Instrukce – x86/AMD64

## Aritmetika – AT&T syntax

operace co, k čemu

addq \$0x05,%rax	$rax = rax + 5$
subl -4(%ebp), %eax	$eax = eax - \text{mem}(ebp-4)$
subl %eax, -4(%ebp)	$\text{mem}(ebp-4) = \text{mem}(ebp-4) - eax$
andX	bitový and – argumenty typu X – b, w, l, q
orX	bitový or
xorX	bitový xor (nejrychlejší vynulování registru)
mulX	násobení čísel bez znamének
divX	dělení čísel bez znamének
imulX	násobení čísel se znaménky
idivX	dělení čísel se znaménky

# Assembler v C programu

```
#include <stdio.h>
int main() {
    int a, b;
    a = -15; b = -5;
    asm volatile (
        "mov %%eax, %%edx;"           ; move value from eax to edx
        "sar $0x1f, %%edx;"          ; clear the sign bit of edx
        "idivl %%ebx; "              ; divide a by b
        : "+a" (a) : "b" (b) : "edx");
    printf ("‐15/(-5)?=%i\n", a);

    a = -15; b = -5;
    asm volatile (
        "xor %%edx, %%edx;"          ; clear edx
        "divl %%ebx;"                ; divide a by b
        : "+a" (a) : "b" (b) : "edx" );
    printf ("‐15/(-5)?=%i\n", a);
}
```

```
a = -15; b = -5;
asm volatile (
    "mov %%eax, %%edx;"           ; move value from eax to edx
    "sar $0x1f, %%edx;"          ; clear the sign bit of edx
    "idivl %%ebx; "              ; divide a by b
    : "+a" (a) : "b" (b) : "edx" );
printf ("‐15/(-5)?=%i\n", a);
return 0;
```

# Instrukce – x86/AMD64

Aritmetika s jedním operandem – AT&T syntax

operace s cim

incl %eax      eax = eax + 1

decw (%ebx)    mem(ebx) = mem(ebx)-1

shlb \$3, %al    al = al«3

shrb \$1, %bl    bl=11000000, po bl=01100000

sarb \$1, %bl    bl=11000000, po bl=11100000

rорx, rolx     bitová rotace doprava a doleva

rcrx, rcl        bitova rotace – pres C – carry flag

# Instrukce – x86/AMD64

## Podmíněné skoky

test a1, a2    tmp = a1 AND a2, Z tmp=0, C tmp<0

cmp a1, a2    tmp = a1-a2, Z tmp=0, C tmp<0

pak lze použít následující skoky

jmp kam                nepodmíněný skok, vlastně %eip=kam

je kam                jmp equal – skoč při rovnosti

jne kam                jmp not equal – skoč při nerovnosti

jg/ja kam                jmp greater – skoč pokud je a1 > a2 (sign/unsig)

jge/jae kam                skoč pokud je a1  $\geq$  a2 (sign/unsig)

jl/jb kam                jmp less – skoč pokud je a1 < a2 (sign/unsig)

jle/jbe kam                skoč pokud je a1  $\leq$  a2 (sign/unsig)

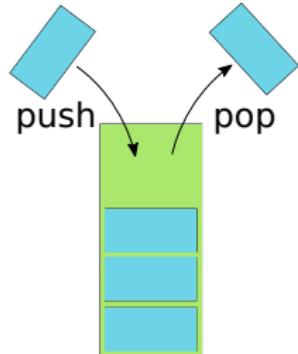
jz/jnz kam                skoč pokud je Z=1/0

jo/jno kam                skoč pokud je O (overflow) = 1/0

# Zásobník

## Zásobník:

- obecná struktura LIFO
- operace push vloží data do zásobníku
- operace pop vybere data ze zásobníku



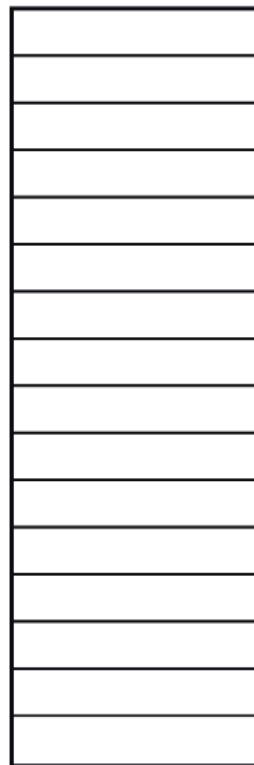
## Implementace:

- implementace registrem *SP* - ukazuje na vrchol zásobníku
- konvence - při každém pop se zvětšuje registr *SP* o velikost operandu, při push se *SP* zmenšuje.

<code>pushl %eax</code>	ulož eax na zásobník
<code>popw %bx</code>	vyber ze zásobníku 2 bajty do bx
<code>pushf/popf</code>	ulož/vyber register EFLAGS
<code>pusha/popa</code>	ulož/vyber všechny uživatelské registry

# Zásobník

push %eax:



pop %eax:

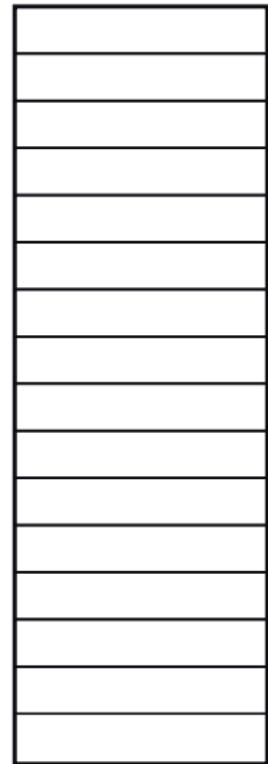
## Zásobník - příklad

```
#include <stdio.h>
#include <unistd.h>
int main();
int b() {
    printf ("Co\u20actu\u20acdelam?\n");
    return 1;
}
void f(int x) {
    unsigned int local[2];
    int i;
    local [0]=256;
    local [1]=257;
    for (i=10; i>=0; i--) {
        printf ("%02i\u20ac\u20ac%08x\n", i, local[i]);
    }
    local [7]=11;
    local [6]=((unsigned int)&b)+3;
    printf ("x=%i\u20acmain\u20ac%p\n", x, &main);
}
int main() {
    f(10);
    printf ("Proc?\n");
    return 0;
}
```

```
push %ebp  
mov %esp,%ebp  
push %ebx  
sub $0x14,%esp
```

**leave**  
**ret**

```
push $0xa  
call 57c <f>
```



# Funkce zásobníku

Zásobník:

- parametry pro funkci
- kam se vrátit po ukončení funkce, místo odkud program volal funkci
- lokální proměnné funkce
  - zásobník je většinou malý
  - omezená velikost lokálních proměnných
  - pozor při rekurzi - lépe se rekurzi vyhnout

# Instrukce – x86/AMD64

Volání funkce

```
call adr    vlastně push %eip, jmp adr
ret         vlastně pop %eip
leave       vlastně mov %ebp, %esp, pop %ebp
```

Lokální proměnné ve funkci – příklad implementace

```
push %ebp      ; Uložíme hodnotu EBP do zasobníku
mov %esp, %ebp ; Zkopirujeme hodnotu registru ESP do EBP
sub $12, %esp  ; Snížíme ukazatel zasobníku o 3x4 bajty
```

První proměnná bude na adrese -4(%ebp), druhá -8(%ebp)

První parametr bude na adrese 8(%ebp), další 12(%ebp)

```
mov %ebp, %esp ; Vratíme ukazatel zpět na původní pozici.
pop %ebp       ; Obnovíme původní hodnotu registru EBP
ret             ; Navrat z funkce
```

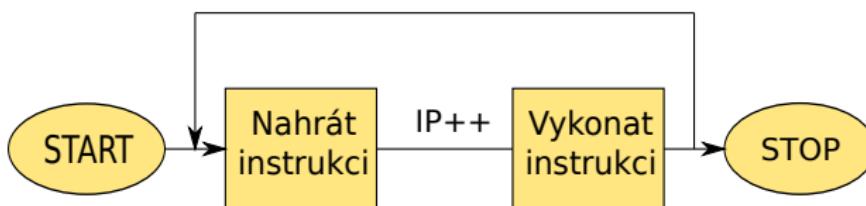
# Instrukce – x86/AMD64

## Složitost assembleru

- Algoritmus se dá přeložit různými způsoby do assembleru
- Různé způsoby pracují různě rychle a jsou rozdílně dlouhé a rozdílně přehledné
- xor %ebx, %ebx je to samé jako mov \$0, %ebx
- lea adresa, registr – load effective address – nastaví hodnotu ukazatele do zadáного registru
- lea -12(%esp), %esp je to samé jako sub \$12, %esp
- lea je výhodnější vzhledem k předzpracování instrukcí, nezatěžuje ALU jednotku (ovšem třeba Atom má zpracování adr. pomalejší než ALU).

# Pracovní krok procesoru

- Procesor pracuje v krocích.
- Jeden krok obsahuje fáze:
  - Přípravná fáze (fetch cycle)
    - nahrává do procesoru instrukci podle IP a umístí její kód do IR
    - na jejím konci se inkrementuje IP
  - Výkonná fáze (execute cycle)
    - vlastní provedení instrukce
    - může se dále obracet (i několikrát) k paměti

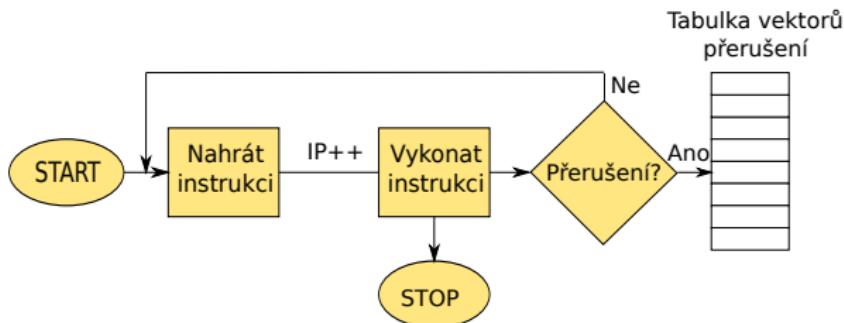


```
loop: FETCH;      /* z adresy IP nahraj data do IR */  
    Increment(IP);  
    EXECUTE;     /* provede operaci uloženou v IR */  
end loop
```

# Přerušení (výjimky)

- Přerušení normální posloupnosti provádění instrukcí
  - cílem je zlepšení účinnosti práce systému
  - je potřeba provést jinou posloupnost příkazů jako reakci na nějakou „neobvyklou“ externí událost
  - přerušující událost způsobí, že se pozastaví běh aktuálně vykonávaného programu v CPU takovým způsobem, aby ho bylo možné později znova obnovit, aniž by to přerušený program „poznał“
- Souběh I/O operace
  - přerušení umožní, aby po začátku přenosu dat z/do periférie CPU prováděla jiné akce než čekání na konec I/O operace
  - činnost CPU se později přeruší iniciativou „I/O modulu“
  - CPU předá řízení na obslužnou rutinu přerušení (Interrupt Service Routine) – standardní součást OS
- CPU testuje nutnost věnovat se obsluze přerušení alespoň po dokončení každé instrukce
  - existují výjimky (např. „blokové instrukce“ Intel)

# Pracovní krok s přerušením



```

INTF=False; /* vymaz preruseni */
loop: FETCH;
  Increment(IP);
  EXECUTE;
  IF povoleno preruseni && INTF then
    Uloz FLAGS na zasobnik
    Uloz IP na zasobnik
    FLAGS nastav CPL0 a zakaz preruseni
    IP = vektoru preruseni
  end loop
  
```

# Obsluha přerušení

- Žádost se vyhodnotí na přípustnost (priority přerušení)
- Procesor přejde do zvláštního cyklu
  - FLAGS se uloží na zásobník (registrovým hodnotám se mění již při vstupu do přerušení a také většina instrukcí mění hodnotu FLAGS; je tedy nutné ho uložit co nejdříve).
  - Na zásobník se uloží i hodnota čítače instrukcí IP (návratová hodnota z přerušení).
  - Do FLAGS se vygeneruje nové stavové slovo s nastaveným CPL0. Nyní je CPU v privilegovaném režimu
  - IP se nahradí hodnotou z vektoru přerušení – skok na obsluhu přerušení
- Procesor přechází do normálního režimu práce a zpracovává obslužnou rutinu přerušení v privilegovaném módu
  - Obslužná rutina musí být transparentní, tj. programově se musí uložit všechny registry CPU, které obslužná rutina použije, a před návratem z přerušení se opět vše musí obnovit tak, aby přerušená posloupnost instrukcí nepoznala, že byla přerušena.
  - Obslužnou rutinu končí instrukce „návrat z přerušení“ IRET mající opačný efekt: z vrcholu zásobníku vezme položky, které umístí zpět do IP a FLAGS

# Druhy přerušení (x86)

- Každé přerušení má své číslo odkazující do tabulky přerušení, kde je tzv. vektor přerušení
- Vektor přerušení obsahuje adresu programu, od které se začně vykonávat kód při výskytu daného přerušení
- Přerušení se dělí vzhledem k vykonávanému programu na synchronní a asynchronní

## Synchronní přerušení

- Chyba dělení (dělení nulou) 0
- Program break 3
- Chybná instrukce 6
- Chybějící segment 11
- Chyba segmentu zásobníku 12
- Chyba ochrany 13
- Chyba stránky 14

## Asynchronní přerušení

- Nemaskovatelné přerušení 2
- časovač 32
- uživatelské přerušení 32–255 (sítová karta, klávesnice, ...)

# Zdroje přerušení

- Vnitřní přerušení – problém při zpracování strojové instrukce
  - instrukce nebo data nejsou v paměti - chyba stránky, chyba segmentu
  - instrukci nelze provést - dělení nulou, ochrana paměti, nelegální instrukce
  - nutno reagovat okamžitě, nelze dokončit instrukci, někdy nelze ani načíst instrukci
- Vnější přerušení – vstupně/výstupní zařízení
  - asynchronní s během procesoru
  - signalizace potřeby reagovat na vstup/výstup
  - reakce po dokončení vykonávané instrukce
- Programové přerušení – strojová instrukce provede' přerušení
  - využívá se k ochraně jádra OS
  - obsluha přerušení může používat privilegované instrukce
  - lze spustit pouze kód připravený OS

# Vícenásobné přerušení

- Sekvenční zpracování
  - během obsluhy jednoho přerušení se další požadavky nepřijímají (pozdržují se, IF bit v registru FLAGS)
  - jednoduché, ale nevhodné pro časově kritické akce
- Vnořené zpracování
  - prioritní mechanismus
  - přijímají se přerušení s prioritou striktně vyšší, než je priorita obsluhovaného přerušení
- Odložené zpracování
  - V přerušení se provede pouze nejnutnější obsluha zařízení, zbytek se provede později mimo přerušení (deffered jobs, workqueues, ...)
  - Neblokují se zbytečně další přerušení

# B4B35OSY: Operační systémy

## Lekce 2. Systémové volání

Petr Štěpán

[stepan@fel.cvut.cz](mailto:stepan@fel.cvut.cz)



10. září, 2020

# Outline

1 Složení OS

2 Služby OS

3 Struktura OS

4 Procesy

# Obsah

1 Složení OS

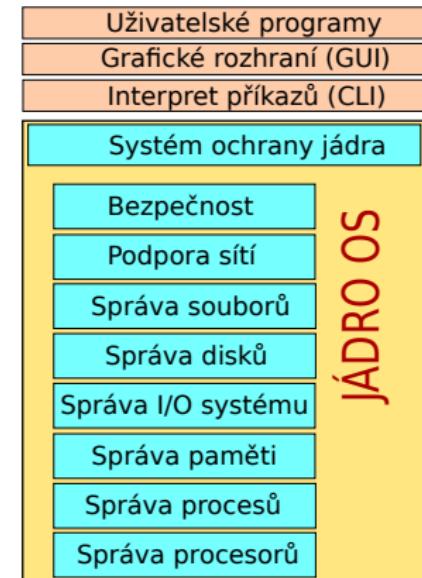
2 Služby OS

3 Struktura OS

4 Procesy

# Složky OS

- Správa procesorů
- Správa procesů
- Správa (hlavní, vnitřní) paměti
- Správa I/O systému
- Správa disků – vnější (sekundární) paměti
- Správa souborů
- Podpora sítí
- Bezpečnost - security
- Systém ochrany jádra



# Interpret příkazů

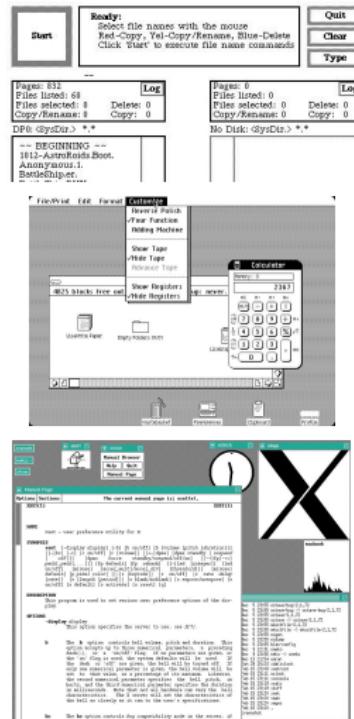
- Většina zadání uživatele je předávána operačnímu systému řídícími příkazy, které zadávají požadavky na
  - správu a vytváření procesů
  - ovládání I/O
  - správu sekundárních pamětí
  - správu hlavní paměti
  - zpřístupňování souborů
  - komunikaci mezi procesy
  - práci v síti, ...
- Program, který čte a interpretuje řídicí příkazy se označuje v různých OS různými názvy
  - Command-line interpreter (CLI), shell, cmd.exe, sh, bash, ...
  - Většinou rozumí jazyku pro programování dávek (tzv. skriptů)
  - Interpret příkazů není částí jádra OS
  - Interpret příkazů pracuje v uživatelském režimu, který je stejný jako pro Vaše programy

## Systémové nástroje

- Poskytují prostředí pro vývoj a provádění programů
- Typická skladba
  - Práce se soubory, editace, kopírování, katalogizace, ...
  - Získávání, definování a údržba systémových informací
  - Modifikace souborů
  - Podpora prostředí pro různé programovací jazyky
  - Sestavování programů
  - Komunikace
  - Anti-virové programy
  - Šifrování a bezpečnost
  - Aplikační programy z různých oblastí
- Systémové nástroje pracují v uživatelském režimu, který je stejný jako pro Vaše programy

# GUI

- První Xerox Alto (1973)
- Apple Lisa (1983)
- X window (1984) – MIT, možnost vzdáleného terminálu přes síť
- Windows 1.0 pro DOS (1985)
- Windows 3.1 (1992) podpora 32-bitových procesorů s ochranou paměti, vylepšená grafika
- Windows NT (1993) – preemptivní multitasking, předchůdce Windows XP (2001)



# Jádro OS

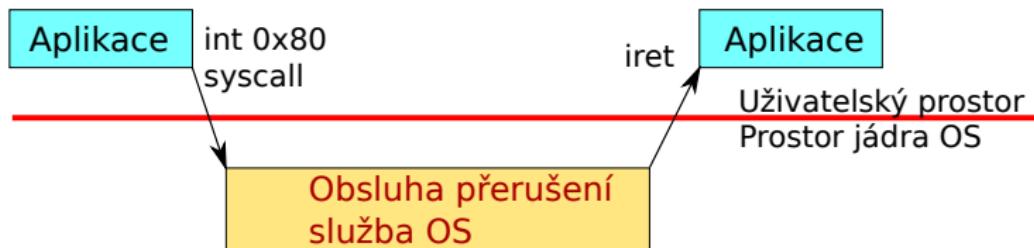
- Poskytuje ochranu/izolaci
  - Aplikačních programů mezi sebou
  - Hardwaru před škodlivými aplikacemi
  - Dat (souborů) před neoprávněnou manipulací
- Řídí přidělování zdrojů aplikacím
  - Paměť, procesorový čas, přístup k HW, síti, ...
- Poskytuje aplikacím služby
  - Jaké?

# Ochrana jádra OS

- Ochrana jádra
  - mechanismus pro kontrolu a řízení přístupu k systémovým a uživatelským zdrojům (paměť, HW zařízení, soubory, ...)
- Systém ochran „prorůstá“ všechny vrstvy OS
- Systém ochran musí
  - rozlišovat mezi autorizovaným a neautorizovaným použitím
  - poskytnout prostředky pro prosazení legální práce
- Detekce chyb
  - Chyby interního a externího hardware
    - Chyby paměti, výpadek napájení
    - Chyby na vstupně/výstupních zařízeních či mediích („díra“ na disku)
  - Softwarové chyby
    - Aritmetické přetečení, dělení nulou
    - Pokus o přístup k „zakázaným“ paměťovým lokacím (ochrana paměti)
- OS nemůže obsloužit žádost aplikačního programu o službu
  - Např. „k požadovanému souboru nemáš právo přistupovat“

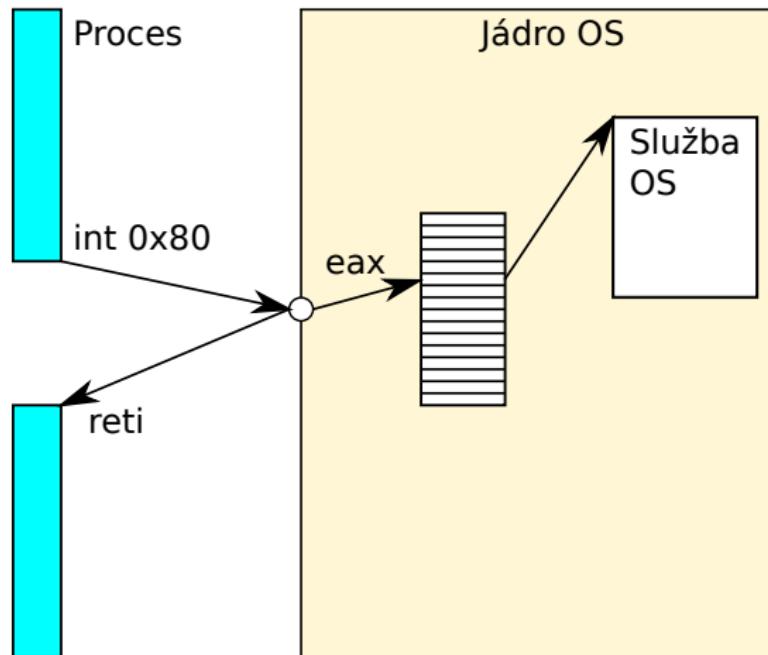
# Ochrana jádra OS

- Základ ochrany OS, přechod do systémového módu
  - Intel x86 rozlišuje 4 úrovně ochrany (priviledge level): 0 – jádro OS, 3 – uživatelský mód
  - Jiné architektury mají většinou jen dva módy (jeden bit ve stavovém slově)
  - V uživatelském módu jsou některé instrukce zakázané (opakování – jaké?)
- Přechod z uživatelského módu do systémového
  - pouze programově vyvolaným přerušením
  - speciální instrukce (trap, int, sysenter, swi, ...)
  - nejde spustit cokoliv, spustí se pouze kód připravený operačním systémem
  - Systémová volání – služby jádra (system calls)
- Přechod ze systémového módu do uživatelského:
  - Speciální instrukce či nastavení odpovídajících bitů ve stavovém slově FLAGS
  - Návrat z přerušení



# Ochrana jádra OS

- Uživatel má do jádra OS přístup pouze přes obsluhu přerušení



# Obsah

1 Složení OS

2 Služby OS

3 Struktura OS

4 Procesy

# Služby jádra OS

## x86 System Call Example – Hello World on Linux

```
.section .rodata
greeting:
.string "Hello World\n"

.text
.global _start
_start:
    mov $4,%eax          ; write is syscall no. 4
    mov $1,%ebx          ; file descriptor, 1 je stdout
    mov $greeting,%ecx   ; address of the data
    mov $12,%edx         ; length of the data
    int $0x80             ; call the system
```

# Služby jádra OS

- Služby jádra jsou číslovány
  - Registr eax obsahuje číslo požadované služby
  - Ostatní registry obsahují parametry, nebo odkazy na parametry
  - Problém je přenos dat mezi pamětí jádra a uživatelským prostorem
    - malá data lze přenést v registrech – návratová hodnota funkce
    - velká data – uživatel musí připravit prostor, jádro z/do něj nakopíruje data, předává se pouze adresa (ukazatel)
- Volání služby jádra na strojové úrovni není komfortní
  - Je nutné použít assembler, musí být dodržena volací konvence
  - Zapouzdření pro programovací jazyky – API
  - Základem je běhová knihovna jazyka C (libc, C run-time library)
- Linux system call table  
[http://docs.cs.up.ac.za/programming/asm/derick\\_tut/syscalls.html](http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html)
- Windows system call table  
<http://j00ru.vexillium.org/ntapi/>

# Application Binary Interface – ABI

- Definuje rozhraní na úrovni strojového kódu:
  - V jakých registrech se předávají parametry
  - V jakém stavu je zásobník
  - Zarovnání vícebytových hodnot v paměti
- ABI se liší nejen mezi OS, ale i mezi procesorovými architekturami stejného OS.
  - Např: Linux i386, amd64, arm, ...
  - Možnost podpory více ABI: int 0x80, sysenter, 32/64 bit

# ABI Linuxu

32 bitový systém (i386):

instrukce `int 0x80`

EIP a EFLAGS se ukládají na zásobník

Popis	Registr
číslo syscall	eax
první argument	ebx
druhý argument	ecx
třetí argument	edx
čtvrtý argument	esi
pátý argument	edi
šestý argument	ebp

64 bitový systém (amd64):

instrukce `syscall`

rychlejší přechod do jádra OS,  
RIP a RFLAGS ukládá do registrů RCX a R11

Popis	Registr
číslo syscall	rax
první argument	rdi
druhý argument	rsi
třetí argument	rdx
čtvrtý argument	r10
pátý argument	r9
šestý argument	r8

# Application Programming Interface – API

- Definice rozhraní pro služby OS (system calls) na úrovni zdrojového kódu
  - Jména funkcí, parametry, návratové hodnoty, datové typy
- POSIX (IEEE 1003.1, ISO/IEC 9945)
  - Specifikuje nejen system calls ale i rozhraní standardních knihovních podprogramů a dokonce i povinné systémové programy a jejich funkcionalitu (např. ls vypíše obsah adresáře)
  - <http://www.opengroup.org/onlinepubs/9699919799/nframe.html>
- Win API
  - Specifikace volání základních služeb systému v MS Windows
- Nesystémová API:
  - Standard Template Library pro C++
  - Java API
  - REST API webových služeb

# Volání služeb jádra OS přes API

Aplikační program (proces) volá službu OS:

- Zavolá podprogram ze standardní systémové knihovny
- Ten transformuje volání na systémové ABI a vykoná instrukci pro systémové volání
- Ta přepne CPU do privilegovaného režimu a předá řízení do vstupního bodu jádra
- Podle kódu požadované služby jádro zavolá funkci implementující danou službu (tabulka ukazatelů)
- Po provedení služby se řízení vrací aplikaci programu s případnou indikací úspěšnosti

# POSIX

- Portable Operating System Interface for Unix – IEEE standard pro systémová volání i systémové programy
- Standardizační proces začal 1985 – důležité pro přenos programů mezi systémy
- 1988 POSIX 1 Core services – služby jádra
- 1992 POSIX 2 Shell and utilities – systémové programy a nástroje
- 1993 POSIX 1b Real-time extension – rozšíření pro operace reálného času
- 1995 POSIX 1c Thread extension – rozšíření o vlákna
- Po roce 1997 se spojil s ISO a byl vytvořen standard POSIX:2001 a POSIX:2008

# UNIX

- Operační systém vyvinutý v 70. letech v Bellových laboratořích
- Protiklad tehdejšího OS Multix
- Motto: **V jednoduchosti je krása**
- Ken Thompson, Dennis Ritchie
- Pro psaní OS si vyvinuli programovací jazyk C
- Jak UNIX tak C přežilo do dnešních let
- Linux, FreeBSD, \*BSD, GNU Hurd, VxWorks...

# Unix v kostce

- Všechno je soubor<sup>1</sup>
- Systémová volání pro práci se soubory:
  - open(pathname, flags) → file descriptor (celé číslo)
  - read(fd, data, délka)
  - write(fd, data, délka)
  - ioctl(fd, request, data) – vše ostatní co není read/write
  - close(fd)
- Souborový systém:
  - /bin – aplikace
  - /etc – konfigurace
  - /dev – přístup k hardwaru
  - /lib – knihovny

---

<sup>1</sup>až na síťová rozhraní, která v době vzniku UNIXu neexistovala

# POSIX dokumentace

- Druhá kapitola manuálových stránek
- Příkaz (např. v Linuxu): man 2 ioctl

ioctl(2) -- Linux man page

## Name

ioctl -- control device

## Synopsis

```
#include <sys/ioctl.h>
int ioctl(int d, int request, ...);
```

## Description

The ioctl() function manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g., terminals) may be controlled with ioctl() requests. The argument d must be an open file descriptor.

The second argument is a device-dependent request code. The third argument is an untyped pointer to memory. It's traditionally char \*argp (from the days before void \* was valid C), and will be so named for this discussion.

# POSIX dokumentace

## Pokračování

An ioctl() request has encoded in it whether the argument is an in parameter or out parameter, and the size of the argument argp in bytes. Macros and defines used in specifying an ioctl() request are located in the file <sys/ioctl.h>.

### Return Value

Usually, on success zero is returned. A few ioctl() requests use the return value as an output parameter and return a nonnegative value on success. On error, -1 is returned, and errno is set appropriately.

### Errors

EBADF d is not a valid descriptor.  
EFAULT argp references an inaccessible memory area.  
EINVAL Request or argp is not valid.  
ENOTTY d is not associated with a character special device.  
ENOTTY The specified request does not apply to the kind of object  
^I that the descriptor d references.

### Notes

In order to use this call, one needs an open file descriptor. Often the open(2) call has unwanted side effects, that can be avoided under Linux by giving it the O\_NONBLOCK flag.

### See Also

`execve(2)`, `fcntl(2)`, `ioctl_list(2)`, `open(2)`, `sd(4)`, `tty(4)`

# Přehled služeb jádra

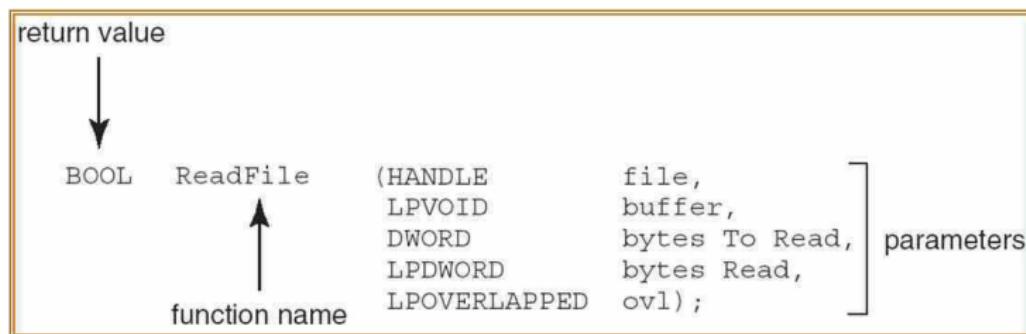
- Práce se soubory
  - open, close, read, write, lseek
- Správa souborů a adresářů
  - mkdir, rmdir, link, unlink, mount, umount, chdir, chmod, stat
- Správa procesů
  - fork, waitpid, execve, exit, kill, signal

# Windows system call API

- Nebylo plně popsáno, skrytá volání využívaná pouze spřátelenými stranami
- Garantováno pouze API poskytované DLL knihovnami (kernel32.dll, user32.dll, ...)
- Win16 – 16 bitová verze rozhraní pro Windows 3.1
- Win32 – 32 bitová verze od Windows NT
- Win32 for 64-bit Windows – 64 bitová verze rozhraní Win32
- Nová window mohou zavést nová volání, případně přečíslovat staré služby.

# Windows API příklad

- Funkce ReadFile() z Win32 API – funkce, která čte z otevřeného souboru



- Parametry předávané funkci ReadFile()
  - HANDLE file – odkaz na soubor, ze kterého se čte
  - LPVOID buffer – odkaz na buffer pro zapsání dat ze souboru
  - DWORD bytesToRead – kolik bajtů se má přečíst
  - LPDWORD bytesRead – kolik bajtů se přečetlo
  - LPOVERLAPPED ovl – zda jde o blokující či asynchronní čtení

# Porovnání POSIX a Win32

<b>POSIX</b>	<b>Win32</b>	<b>Popis</b>
fork	CreateProcess	Vytvoř nový proces
execve	–	CreateProcess = fork + execve
waitpid	WaitForSingleObject	Čeká na dokončení procesu
exit	ExitProcess	Ukončí proces
open	CreateFile	Vytvoří nový soubor nebo otevře existující
close	CloseHandler	Zavře soubor
read	ReadFile	Čte data ze souboru
write	WriteFile	Zapisuje data do souboru
seek	SetFilePointer	Posouvá ukazatel v souboru
stat	GetFileAttributesExt	Vrací informace o souboru
mkdir	CreateDirectory	Vytvoří nový adresář
rmdir	RemoveDirectory	Smaže adresář souborů
link	–	Win32 nepodporuje symbolické odkazy
unlink	DeleteFile	Zruší existující soubor
chdir	SetCurrentDirectory	Změní pracovní adresář

POSIX služby mount, umount, kill, chmod a další nemají ve Win32 přímou obdobu a analogická funkcionality je řešena jiným způsobem.

# Obsah

1 Složení OS

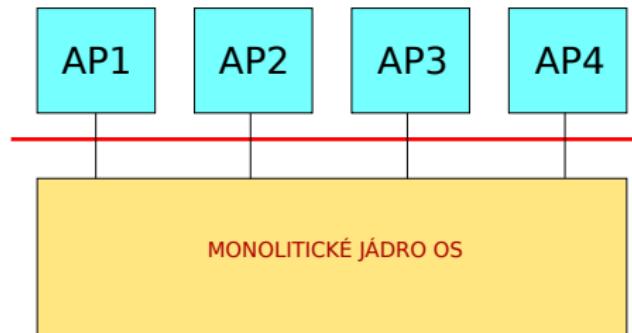
2 Služby OS

3 Struktura OS

4 Procesy

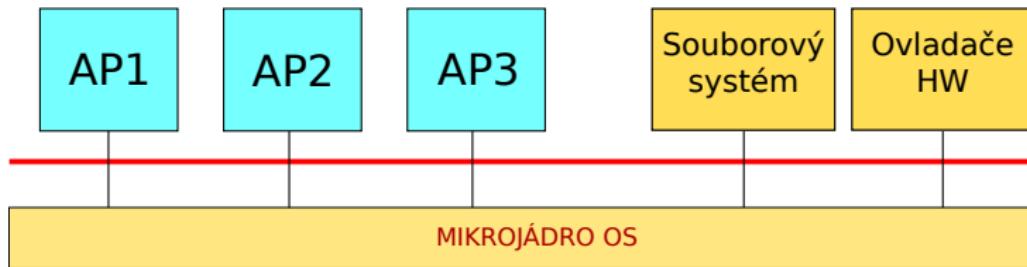
# Vykonání služeb jádra OS

- Klasický monolitický OS
  - Non-process Kernel OS
  - Procesy – jen uživatelské a systémové programy
  - Jádro OS je prováděno jako monolitický (byť velmi složitý) program v privilegovaném režimu
    - „USB MIDI má přístup ke klíči k šifrování disku :-)" CVE-2016-2384
- Služba jádra OS je typicky implementována jako kód v jádře, běžící jako přerušení využívající paměťový prostor volajícího programu

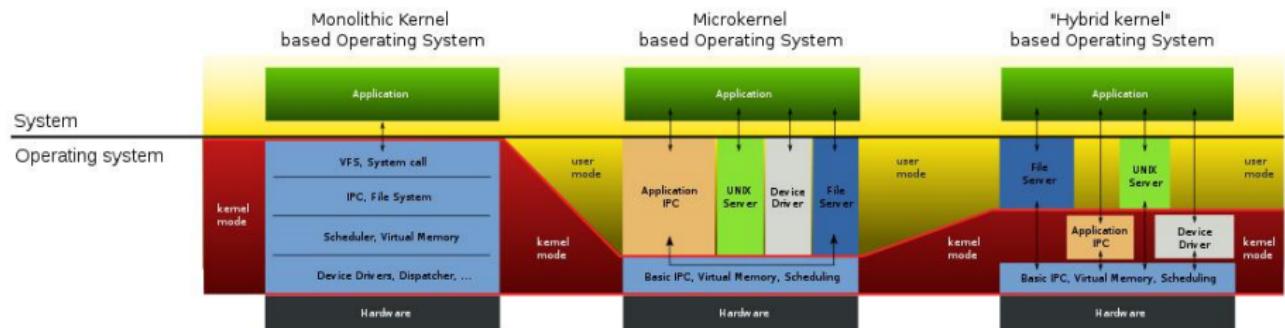


# Procesově orientované jádro OS

- OS je soustavou systémových procesů
- Funkcí jádra je tyto procesy separovat ale umožnit přitom jejich kooperaci
  - Minimum funkcí je potřeba dělat v privilegovaném režimu
  - Jádro pouze ústředna pro přepojování zpráv
  - Řešení snadno implementovatelné i na multiprocesorech
- Malé jádro ⇒ mikrojádro ( $\mu$ -jádro) – (microkernel)



# Porovnání JOS

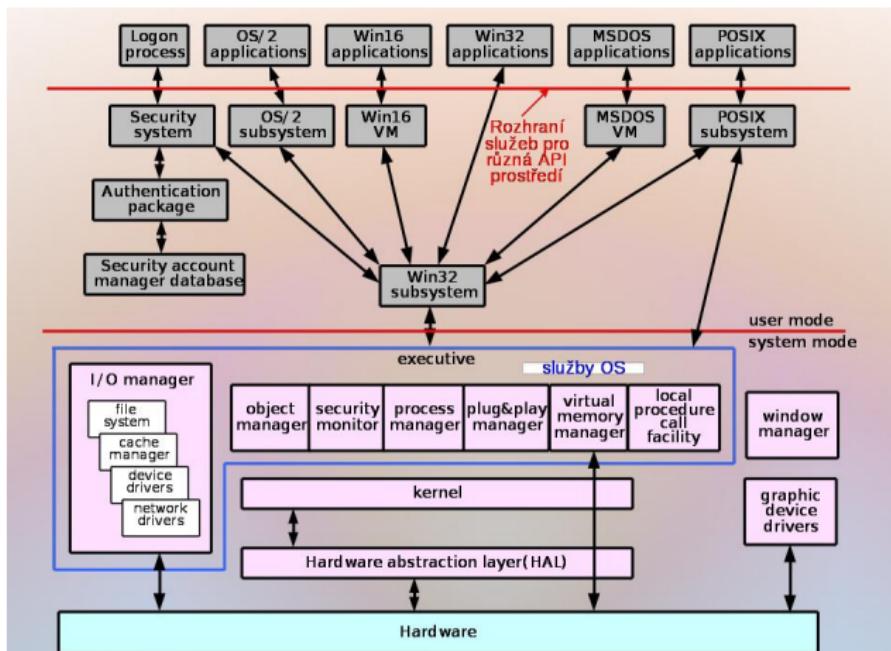


# Mikrojádro – vlastnosti

- OS se snáze přenáší na nové hardware architektury,
  - μ-jádro je malé
- Vyšší spolehlivost – modulární řešení
  - moduly jsou snáze testovatelné
- Vyšší bezpečnost
  - méně kódu se běží v privilegovaném režimu
- Pružnější, snáze rozšiřitelné řešení
  - snadné doplňování nových služeb a rušení nepotřebných
- Služby jsou poskytovány unifikovaně
  - výměnou zpráv
- Přenositelné řešení
  - při implementaci na novou hardware platformu stačí změnit μ-jádro
- Podpora distribuovanosti
  - výměna zpráv je implementována v síti i uvnitř systému
- Podpora objektově-orientovaného přístupu
  - snáze definovatelná rozhraní mezi aplikacemi a μ-jádrem
- To vše za cenu
  - zvýšené režie, volání služeb je nahrazeno výměnou zpráv mezi aplikačními a systémovými procesy

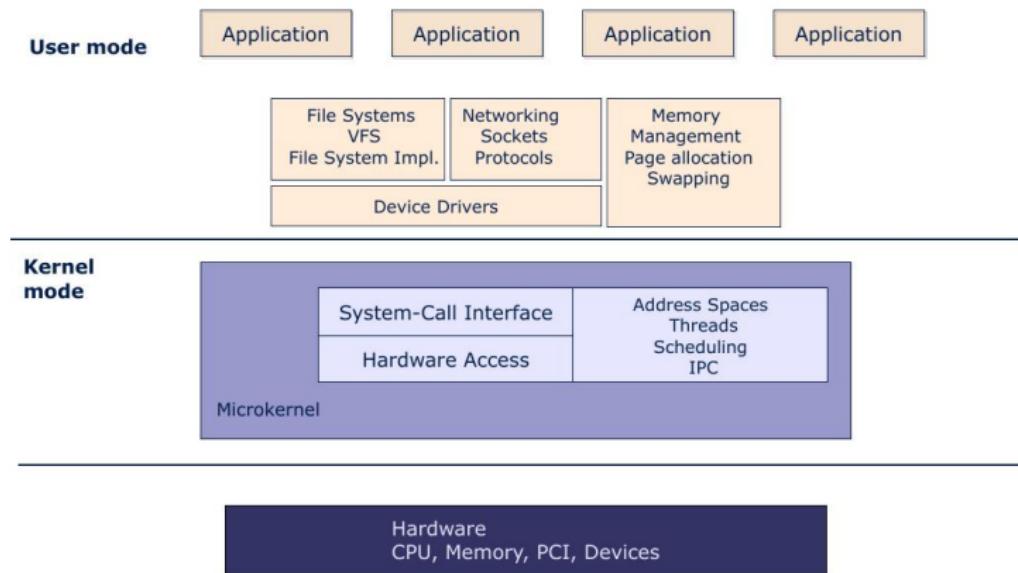
# Windows (XP)

JOS Windows má architekturu  $\mu$ -jádra, ale vše běží v jednom adresním prostoru, takže se jedná o monolitické jádro<sup>2</sup>.



<sup>2</sup> <https://techcommunity.microsoft.com/t5/Windows-Kernel-Internals/One-Windows-Kernel/ba-p/267142>

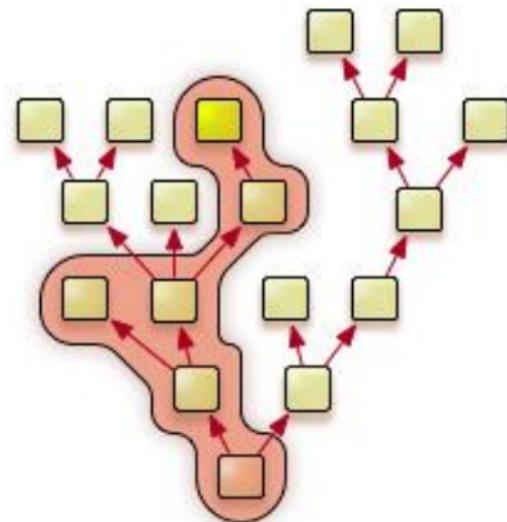
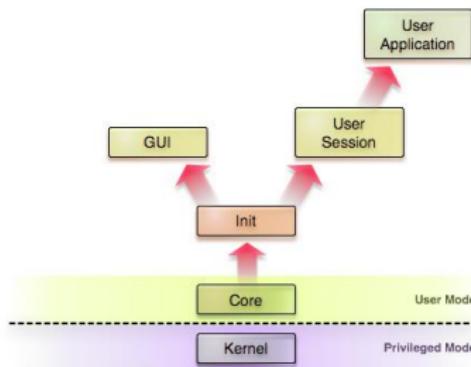
# L4Re – OS se skutečným $\mu$ -jádrem



# Genode – OS se skutečným $\mu$ -jádrem

Jeden z cílů: Omezit velikost "Trusted computing base"

<http://genode.org/>



# NOVA – $\mu$ -jádro

Systémová volání OS NOVA:

- call
- reply
- create\_pd
- create\_ec
- create\_sc
- create\_pt
- create\_sm
- revoke
- lookup
- ec\_ctrl
- sc\_ctrl
- pt\_ctrl
- sm\_ctrl
- assign\_pci
- assign\_gsi

Výukový OS – bude používán na cvičení

- Víc systémových volání opravdu nemá
- PD – protection domain – proces
- EC – execution context
- SC – scheduling context
- PT – portal
- SM – semafor

# Závěr - struktura OS

OS mohou (ale nemusí) být funkčně velmi složité

<b>OS</b>	<b>Rok</b>	<b># služeb</b>
Unix	1971	33
Unix	1979	47
Sun OS4.1	1989	171
4.3 BSD	1991	136
Sun OS4.5	1992	219
Sun OS5.6 (Solaris)	1997	190
WinNT 4.0	1997	3443
Linux 2.0	1998	229
Linux 4.4	2016	332
NOVA	2014	15

Počty cyklů CPU spotřebovaných ve WinXP při

- Zaslání zprávy mezi procesy: 6K–120K (dle použité metody)
- Vytvoření procesu: 3M
- Vytvoření vlákna: 100K
- Vytvoření souboru: 60K
- Vytvoření semaforu: 10K–30K
- Nahrání DLL knihovny" 3M
- Obsluha přerušení/výjimky: 100K–2M
- Přístup do systémové databáze (Registry) : 20K

Počty cyklů CPU spotřebovaných v OS NOVA při

- Zaslání zprávy mezi procesy: 300–600 (dle použité metody)

# Závěr - struktura OS

OS jsou velmi rozsáhlé

Údaje jsou jen orientační, Microsoft data nezveřejňuje

SLoC (Source Lines of Code) je velmi nepřesný údaj: Tentýž programový příkaz lze napsat na jediný nebo celou řadu řádků.

<b>OS</b>	<b>Rok</b>	<b>SLoC</b>
Windows 3.1	1992	3mil.
Windows NT 3.5	1993	4mil.
Windows 95	1995	15mil.
Windows NT 4.0	1997	16mil.
Windows 98 SR-2	1999	18mil.
Windows 2000 SP5	2002	30mil.
Windows XP SP2	2005	48mil.
Windows 7	2010	není známo
Linux 4.13 (jen JOS)	2017	16.8mil.
NOVA	2014	10tis.

# Obsah

1 Složení OS

2 Služby OS

3 Struktura OS

4 Procesy

# Služby OS - procesy

<b>POSIX</b>	<b>Popis</b>
fork	Vytvoří nový proces jako kopii rodičovského
execve	Nahradí běžící process jiným programem - zavede ho do paměti a spustí
waitpid	Čeká na dokončení procesu potomka, přijme výsledek jeho běhu
exit	Ukončí proces, sdělí rodiči výsledek běhu (úspěch/číslo chyby)

# Služby OS - fork, exit

Služba `pid_t fork(void)` vytvoří kopii procesu, která:

- má odlišný PID a rodičovský PID
- má návratovou hodnotu ze systémového volání 0 (rodičovský proces má návratovou hodnotu pid potomka)
- má kopii data a zásobníku

Služba `void exit(int status)`

- ukončí vykonávání procesu
- předá rodiči hodnotu status
- dokud rodič hodnotu nepřečte, tak nelze proces úplně odstranit z paměti

# Příklad fork

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    int f=fork(), ff =-1;
    ff =fork();

    printf ("Hello %i %i\n", f, ff );
    return 0;
}
```

# Příklad fork

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    int f=fork(), ff =-1;
    if (f==0) {
        ff =fork();
    }
    printf ("Hello %i %i\n", f, ff );
    return 0;
}
```

# Služby OS - wait

Služba pid\_t wait(int \*status):

- čeká na ukončení libovolného potomka
- pid\_t wait\_pid(pid\_t pid, int \*status, int opt) čeká na konkrétního potomka
- přijme jeho návratovou hodnotu
- WEXITSTATUS(status) dekóduje 8-bitů od končícího potomka
- WIFEXITED(status) dekóduje, zda potomek skončil normálně - tedy volání služby exit
- WIFSIGNALED(status) dekóduje, zda potomek skončil přijetím signálu
- další makra na detailní zjištění ukončení ptotmka

# Zombie

Pokud potomek skončí a rodičovský proces ještě neskončil a nezavolal systémové volání wait, tak potomek nemůže být odstraněn z tabulky procesů.

Důvod:

- potomek musí předat rodiči výsledek svého běhu
- toto číslo musí být někde uloženo - ve struktuře, která popisuje proces potomka
- potomek nemůže běžet, ale ještě nemůže být úplně ukončen - stav zombie
- viz praktický příklad k přednášce

# Fork bomb

Jednoduchý proces, který sám sebe spustí alespoň dvakrát.

Proces se začne nekontrolovaně množit a hrozí zahlcení systému.

- BASH :() :|:& ;:
  - definice funkce se jménem :
  - funkce : spustí funkci : dvakrát spojenou rourou
  - spustí se první provedení funkce :
- Windows – fork.bat: %0 | %0
  - %0 - obdobně jako v bashi jméno spuštěného programu
  - spusť se dvakrát propojený rourou
- Perl – perl -e "fork while fork"&
- [https://en.wikipedia.org/wiki/Fork\\_bomb](https://en.wikipedia.org/wiki/Fork_bomb)

# B4B35OSY: Operační systémy

## Lekce 3. Procesy a vlákna

Petr Štěpán

[stepan@fel.cvut.cz](mailto:stepan@fel.cvut.cz)



10. září, 2020

# Outline

1 Proces

2 Vlákna

3 Od programu k procesu

# Obsah

1 Proces

2 Vlákna

3 Od programu k procesu

# Proces

- Výpočetní proces (job, task) – spuštěný program
- Proces je identifikovatelný jednoznačným číslem v každém okamžiku své existence
  - PID – Process IDentifier
- Co tvoří proces:
  - Obsahy registrů procesoru (čítač instrukcí, ukazatel zásobníku, příznaky FLAGS, uživatelské registry, FPU registry)
  - Otevřené soubory
  - Použitá paměť:
    - Zásobník – .stack
    - Data – .data
    - Program – .text
- V systémech podporujících vlákna bývá proces chápán jako obal či hostitel svých vláken

# Proces – požadavky na OS

- Umožňovat procesům vytváření a spouštění dalších procesů
- Prokládat – „paralelizovat“ vykonávání jednotlivých procesů s cílem maximálního využití procesoru/ů
- Minimalizovat dobu odezvy procesu prokládáním běhů procesů
- Přidělovat procesům požadované systémové prostředky
  - Soubory, V/V zařízení, synchronizační prostředky
- Umožňovat vzájemnou komunikaci mezi procesy
- Poskytovat aplikačním procesům funkčně bohaté, bezpečné a konzistentní rozhraní k systémovým službám
  - Systémová volání – minulá přednáška

# Vznik procesu

- Rodičovský proces vytváří procesy-potomky
  - pomocí služby OS. Potomci mohou vystupovat v roli rodičů a vytvářet další potomky, ...
  - vzniká tak strom procesů
- Sdílení zdrojů mezi rodiče a potomky:
  - rodič a potomek mohou sdílet všechny zdroje původně vlastněné rodičem (obvyklá situace v POSIXových systémech)
  - potomek může sdílet s rodičem podmnožinu zdrojů rodičem k tomu účelu vyčleněnou
  - potomek a rodič jsou plně samostatné procesy, nesdílí žádný zdroj
- Souběh mezi rodiči a potomky:
  - Možnost 1: rodič čeká na dokončení potomka
  - Možnost 2: rodič a potomek mohou běžet souběžně
- V POSIXových systémech je každý proces potomkem jiného procesu
  - Výjimka: proces č. 1 (init, systemd, ...) vytvořen při spuštění systému
  - Spustí řadu procesů a skriptů (rc), ty inicializují celý systém a vytvoří démony (procesy běžící na pozadí bez přístupu na terminál) service ve Win32
  - Init spustí také
    - textové terminály proces getty, který čeká na uživatele → login → uživatelův shell
    - grafické terminály *display manager* a *greeter* (grafický login)

# POSIX vytvoření procesu – fork/exec

Rodič vytváří nový proces – potomka – voláním služby **fork**

- vznikne identická kopie rodičovského procesu až na:
  - návratovou hodnotu systémového volání
  - hodnotu PID, PPID – číslo rodičovského procesu
- návratová hodnota určuje, kdo je potomek a kdo rodič
  - 0 – jsem potomek
  - PID – jsem rodič a získávám PID potomka
- vytvoří se virtuální kopie programu – fyzicky je vykonáván program shodný s programem rodiče
- potomek může použít volání služby **exec** pro nahradu programu ve svém adresním prostoru jiným programem.
  - Příklad – bash použije funkci fork pro vytvoření potomka, kterého pak nahradí zadaným programem.

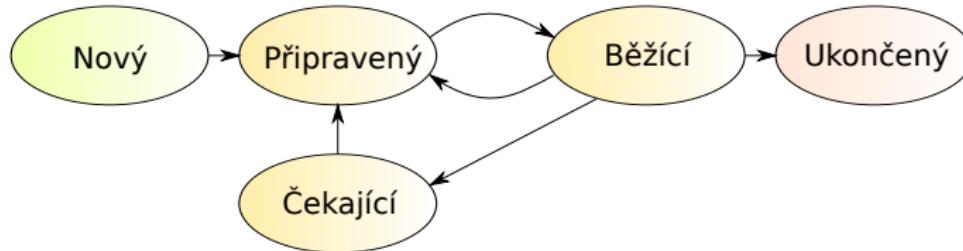
# Ukončení procesu

- Proces provede poslední instrukci programu a žádá OS o ukončení voláním služby `exit(status)`
  - Stavová data procesu-potomka (`status`) se musí předat procesu-rodiči, který typicky čeká na potomka pomocí `wait()`
  - Zdroje (paměť, otevřené soubory) končícího procesu jádro samo uvolní
- Proces může skončit také:
  - přílišným nárokem na paměť (požadované množství paměti není a nebude k dispozici) – *OOM killer* v Linux
  - běžící kód vygeneruje výjimku CPU, kterou jádro neumí vyřešit:
    - aritmetickou chybou (dělení nulou,  $\arcsin(2)$ , ...)
    - pokusem o narušení ochrany paměti („zabloudění“ programu)
    - pokusem o provedení nedovolené (privilegované) operace (zakázaný přístup k hardwarovému prostředku)
    - ...
  - žádostí rodičovského procesu (v POSIXu signál)
    - Může tak docházet ke kaskádnímu ukončování procesů
    - V POSIXu lze proces „odpojit“ od rodiče – démon
  - a v mnoha dalších chybových situacích

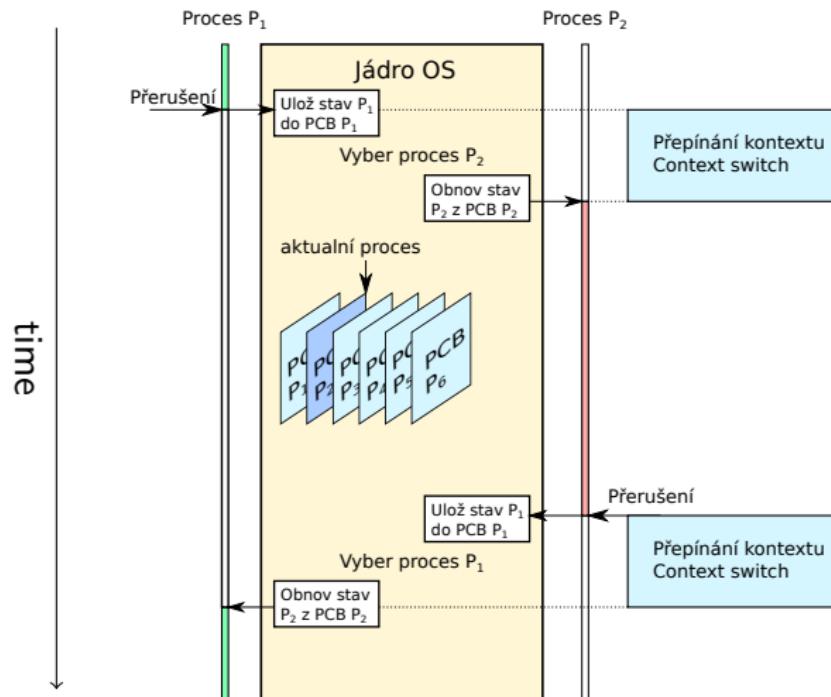
## Stav procesu

Proces se za dobu své existence prochází více stavů a nachází se vždy v jednom z následujících stavů:

- **Nový (new)** – proces je právě vytvářen, ještě není připraven k běhu, ale již jsou připraveny některé části
- **Připravený (ready)** – proces čeká na přidělení procesoru
- **Běžící (running)** – instrukce procesu je právě vykonávány procesorem, tj. interpretovány některým procesorem
- **Čekající (waiting, blocked)** – proces čeká na událost
- **Ukončený (terminated)** – proces ukončil svoji činnost, avšak stále ještě vlastní některé systémové prostředky



# Přepínání procesů



# Přepínání procesů

- Přechod od procesu  $P_1$  k  $P_2$  zahrnuje tzv. přepnutí kontextu
- Přepnutí od jednoho procesu k jinému nastává výhradně v důsledku nějakého přerušení (či výjimky)
- Proces  $P_1$  přejde do jádra operačního systému, který provede přepnutí kontextu → spustí se proces  $P_2$ 
  - Nejprve OS uschová stav původně běžícího procesu  $P_1$  v  $PCB_{P_1}$
  - jádru OS rozhodne, který proces poběží dál –  $P_2$
  - Obnoví se stav procesu  $P_2$  z PCB  $P_2$
- Přepnutí kontextu představuje režijní ztrátu
  - během přepínání systém nedělá nic užitečného, nepoběží žádný proces
  - časově nejnáročnější je správa paměti dotčených procesů
- Doba přepnutí závisí na hardwarové podpoře v procesoru
  - minimální hardwarová podpora je implementace přerušení:
    - uchování IP a FLAGS
    - naplnění IP a FLAGS ze zadaných hodnot
  - lepší podpora:
    - ukládání a obnova více/všech registrů procesoru jedinou instrukcí
    - vytvoří otisk stavu procesoru do paměti a je schopen tento otisk opět načíst (pusha/popa)

# Popis procesů

## Process Control Block (PCB)

- Obsahuje veškeré údaje o procesu – Linux `task_struct` –  
`include/linux/sched.h`
- Datová struktura obsahující:
  - Identifikátor procesu (PID) a rodičovského procesu (PPID)
  - Globální stav (process state)
  - Místo pro uložení všech registrů procesoru
  - Informace potřebné pro plánování procesoru/ů
  - Priorita, historie využití CPU
  - Informace potřebné pro správu paměti
  - Informace o právech procesu, kdo ho spustil
  - Stavové informace o V/V (I/O status)
  - Otevřené soubory
  - Proměnné prostředí (environment variables)
  - … (spousta dalších informací)
  - Ukazatelé pro řazení PCB do front a seznamů

# Fronty procesů pro plánování

- Fronta připravených procesů
  - množina procesů připravených k běhu čekajících pouze na přidělení procesoru
- Fronta na dokončení I/O operace
  - samostatná fronta pro každé zařízení
- Seznam odložených procesů
  - množina procesů čekajících na přidělení místa v hlavní paměti, FAP
- Fronty související se synchronizací procesů
  - množiny procesů čekajících synchronizační události
- Fronta na přidělení prostoru v paměti
  - množina procesů potřebujících zvětšit svůj adresní prostor
- ... (další fronty podle potřeb)
- Procesy mezi různými frontami migrují

# Obsah

1 Proces

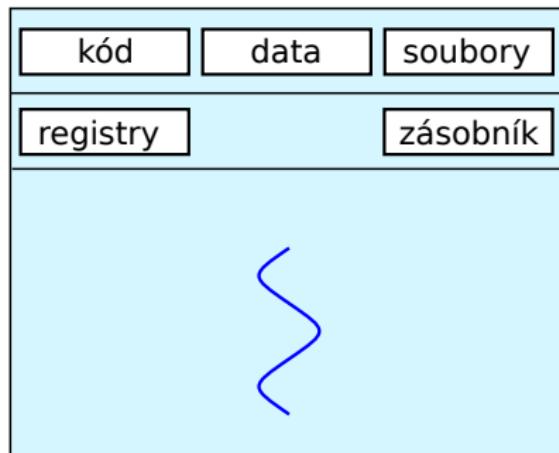
2 Vlákna

3 Od programu k procesu

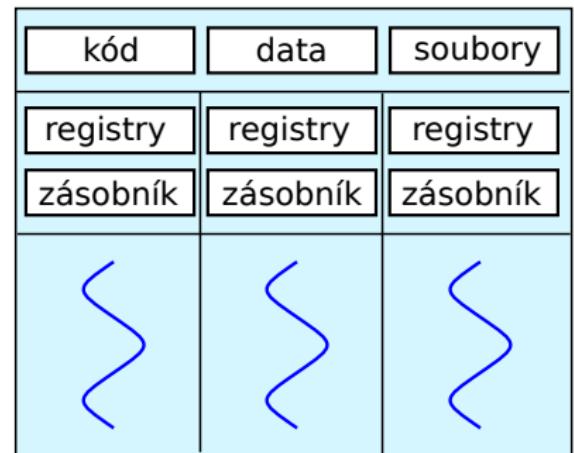
# Program, proces, vlákno

- Program:
  - je soubor (např. na disku) přesně definovaného formátu obsahující
    - instrukce,
    - data
    - údaje potřebné k zavedení do paměti a inicializaci procesu
- Proces:
  - je spuštěný program – objekt jádra operačního systému provádějící výpočet podle programu
  - je charakterizován svým paměťovým prostorem a kontextem (prostor v RAM se přiděluje procesům – nikoli programům!)
  - může vlastnit (kontext obsahuje položky pro) otevřené soubory, I/O zařízení a komunikační kanály, které vedou k jiným procesům, ...
  - obsahuje jedno či více vláken
- Vlákno:
  - je sekvence instrukcí vykonávaných procesorem
  - sdílí s ostatními vlákny procesu paměťový prostor a další atributy procesu (soubory, ...)
  - má vlastní hodnoty registrů CPU

# Procesy a vlákna



jednovláknový proces



vícevláknový proces

# Vlákno – thread

## Vlákno – thread

- Objekt vytvářený v rámci procesu a viditelný uvnitř procesu
- Tradiční proces je proces tvořený jediným vláknem
- Vlákna podléhají plánování a přiděluje se jim strojový čas i procesory
- Vlákno se nachází ve stavech: běží, připravené, čekající, ukončené
- Když vlákno neběží, je kontext vlákna uložený v TCB (Thread Control Block)
  - analogie PCB
    - Linux má stejnou strukturu `task_struct` pro procesy i pro vlákna na informace společné s procesem (např. správa paměti) se vlákno odkazuje k procesu
    - Každý proces je tedy vlastně alespoň jedno vlákno
- Vlákno může přistupovat k globálním proměnným a k ostatním zdrojům svého procesu, data jsou sdílena všemi vlákny stejného procesu
  - Změnu obsahu globálních proměnných procesu vidí všechna ostatní vlákna téhož procesu
  - Soubor otevřený jedním vláknem je viditelný pro všechna ostatní vlákna téhož procesu

# Proces

Co patří komu?

kód programu	proces
lokální proměnné	vlákno
globální proměnné	proces
otevřené soubory	proces
zásobník	vlákno
správa paměti	proces
čítač instrukcí	vlákno
registry CPU	vlákno
plánovací stav	vlákno
uživatelská práva	proces

# Účel vláken

- Přednosti
  - Vlákno se vytvoří i ukončí rychleji než proces
  - Přepínání mezi vlákny je rychlejší než mezi procesy
  - Dosáhne se lepší strukturalizace programu
- Příklady
  - Souborový server v LAN
    - Musí vyřizovat během krátké doby několik požadavků na soubory
    - Pro vyřízení každého požadavku se zřídí samostatné vlákno
  - Symetrický multiprocesor
    - na různých procesorech mohou běžet vlákna souběžně
  - Menu vypisované souběžně se zpracováním prováděným jiným vláknem
    - Překreslování obrazovky souběžně se zpracováním dat
  - Paralelizace algoritmu v multiprocesoru

# Stavy vláken

- Vlákna podléhají plánování a mají své stavy podobně jako procesy
- Základní stavy
  - běžící
  - připravené
  - čekající
- Všechna vlákna jednoho procesu sdílejí společný adresní prostor
- Vlákna se samostatně neodkládají na disk(process swap), odkládá je jen proces
- Ukončení (havárie) procesu ukončuje všechna vlákna existující v tomto procesu

# Možnosti implementace vláken

## Vlákna na uživatelské úrovni

- OS zná jenom procesy
- Vlákna vytváří uživatelská knihovna, která střídavě mění spuštěná vlákna procesu
- Pokud jedno vlákno na něco čeká, ostatní vlákna nemohou běžet, protože jádro OS označí jako čekající celý proces
- Pouze staré systémy, nebo jednoduché OS, kde nejsou vlákna potřeba

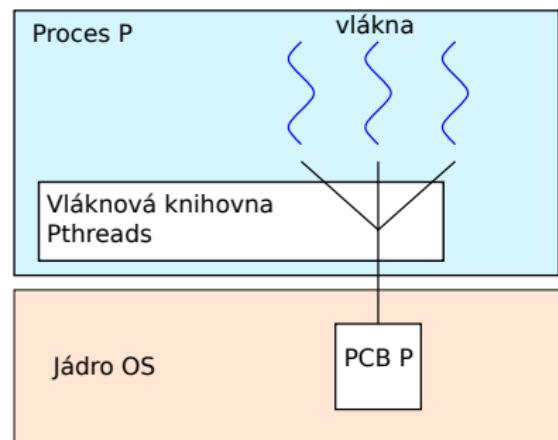
## Vlákna na úrovni jádra OS

- Procesy a vlákna jsou plně podporované v jádře
- Moderní operační systémy (Windows, Linux, OSX, Android)
- Vlákno je jednotka plánování činnosti systému

# Vlákna na uživatelské úrovni

## Problémy vláken na uživatelské úrovni

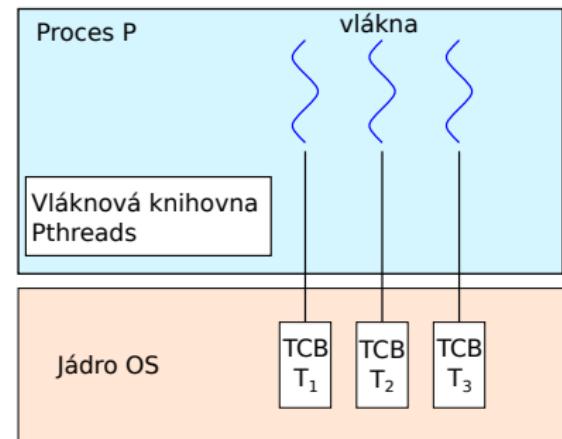
- Jedno vlákno čeká, všechny vlákna čekají
- Proces čeká, ale stav vlákna je běžící
- Dvě vlákna nemohou běžet skutečně paralelně, i když systém obsahuje více CPU



# Vlákna v jádře OS

## Kernel-Level Threads (KLT)

- Veškerá správa vláken je realizována OS
- Každé vlákno v uživatelském prostoru je zobrazeno na vlákno v jádře (model 1:1)
- JOS vytváří, plánuje a ruší vlákna
- Jádro může plánovat vlákna na různé CPU, skutečný multiprocessing
- Nyní všechny moderní OS: Windows, OSX, Linux, Android



# Vlákna v jádře OS

- Výhody:
  - Volání systému neblokuje ostatní vlákna téhož procesu
  - Jeden proces může využít více procesorů
  - Skutečný paralelismus uvnitř jednoho procesu – každé vlákno běží na jiném procesoru
  - Tvorba, rušení a přepínání mezi vlákny je levnější než přepínání mezi procesy
  - Netřeba dělat cokoliv s přidělenou pamětí
- Nevýhody:
  - Systémová správa je režijně nákladnější než u čistě uživatelských vláken
  - Klasické plánování není "spravedlivé": Dostává-li vlákno své časové kvantum, pak procesy s více vlákny dostávají více času
    - Moderní OS ale používají spravedlivé plánování

# Pthreads

- Pthreads je knihovna poskytující API pro vytváření a synchronizaci vláken definovaná standardem POSIX.
- Knihovna Pthreads poskytuje unifikované API:
  - Nepodporuje-li JOS vlákna, knihovna Pthreads bude pracovat čistě s ULT
  - Implementuje-li příslušné jádro KLT, pak toho knihovna Pthreads bude využívat
  - Pthreads je tedy systémově závislá knihovna
- Vlákna Linux:
  - Linux nazývá vlákna tasks
  - Linux má stejnou strukturu task\_struct pro procesy i pro vlákna
  - Lze použít knihovnu pthreads
  - Vytváření vláken je realizováno službou OS clone()

# PThreads API

Příklad: Samostatné vlákno, které počítá součet prvních n celých čísel

```
#include <pthread.h>
#include <stdio.h>

int sum;                                /* sdílená data */
void *runner(void *param);                /* rutina realizující vlákno */
main(int argc, char *argv[]) {
    pthread_t tid;                      /* identifikátor vlákna*/
    pthread_attr_t attr;                 /* atributy vlákna */
    pthread_attr_init(&attr);           /* inicializuj implicitní atributy */
    pthread_create(&tid, &attr, runner, argv[1]); /* vytvoř vlákno */
    pthread_join(tid,NULL);             /* čekej až vlákno skončí */
    printf("sum = %d\n",sum);
}

void *runner(void *param) {
    int upper = atoi(param);  int i;  sum = 0;
    if (upper > 0) {
        for (i = 1; i <= upper; i++)
            sum += i;
    }
    pthread_exit(0);
}
```

# Vlákna ve Windows

- Aplikace ve Windows běží jako proces tvořený jedním nebo více vlákny
- Windows implementují mapování 1:1
- Některí autoři dokonce tvrdí, že Proces se nemůže vykonávat, neboť je jen kontejnerem pro vlákna a jen ta jsou schopná běhu
- Každé vlákno má:
  - svůj identifikátor vlákna
  - sadu registrů (obsah je ukládán v TCM)
  - samostatný uživatelský a systémový zásobník
  - privátní datovou oblast

# Vlákna v Javě

## ■ Vlákna v Javě:

- Java má třídu „Thread“ a instancí je vlákno
- Samozřejmě lze z třídy Thread odvodit podtřídu a některé metody přepsat
- JVM pro každé vlákno vytváří jeho Java zásobník, kde jsou lokální třídy nedostupné pro ostatní vlákna
- JVM spolu se základními Java třídami vlastně vytváří virtuální stroj obsahující jak vlastní JVM tak i na něm běžící OS podporující vlákna
- Pokud se jedná o OS podporující vlákna pak jsou vlákna JVM mapována 1:1 na vlákna OS

# Vlákna v Javě

Dva příklady jak vytvořit vlákno v Javě

```
class CounterThread extends Thread {  
    public void run() {  
        for(int i = 0; i < 10; i++) {  
            System.out.println(i);  
        }  
    }  
}
```

```
Thread counterThread = new CounterThread();  
  
counterThread.start();
```

```
class Counter implements Runnable {  
    public void run() {  
        for(int i = 0; i < 10; i++) {  
            System.out.println(i);  
        }  
    }  
}
```

```
Runnable counter = new Counter();  
Thread counterThread = new Thread(counter);  
  
counterThread.start();
```

# Obsah

1 Proces

2 Vlákna

3 Od programu k procesu

# Psaní programů

- Psaní programu je prvním krokem po analýze zadání a volbě algoritmu
- Program zpravidla vytváříme textovým editorem a ukládáme do souboru s příponou indikující programovací jazyk
  - zdroj.c pro jazyk C
  - prog.java pro jazyk Java
  - text.cc, text.cpp pro C++
- Každý takový soubor obsahuje úsek programu označovaný dále jako modul
- V závislosti na typu dalšího zpracování pak tyto moduly podléhají různým sekvencím akcí, na jejichž konci je jeden nebo několik výpočetních procesů
- Rozlišujeme dva základní typy zpracování:
  - Interpretace (bash, python)
  - Kompilace – překlad (C, Pascal)
  - existuje i řada smíšených přístupů (Java – vykonává předkompilovaný a uložený kód, vytvořený překladačem, který je součástí interpretačního systému)

# Interpretace

Interpretem rozumíme program, který provádí příkazy napsané v nějakém programovacím jazyku

- vykonává přímo zdrojový kód
  - mnohé skriptovací jazyky a nástroje (např. bash), starší verze BASIC
- překládá zdrojový kód do efektivnější vnitřní reprezentace a tu pak okamžitě „vykonává“
  - jazyky typu Perl, Python, MATLAB apod.

Výhody interpretů:

- rychlý vývoj bez potřeby explicitního překladu a dalších akcí
- nezávislost na cílovém stroji

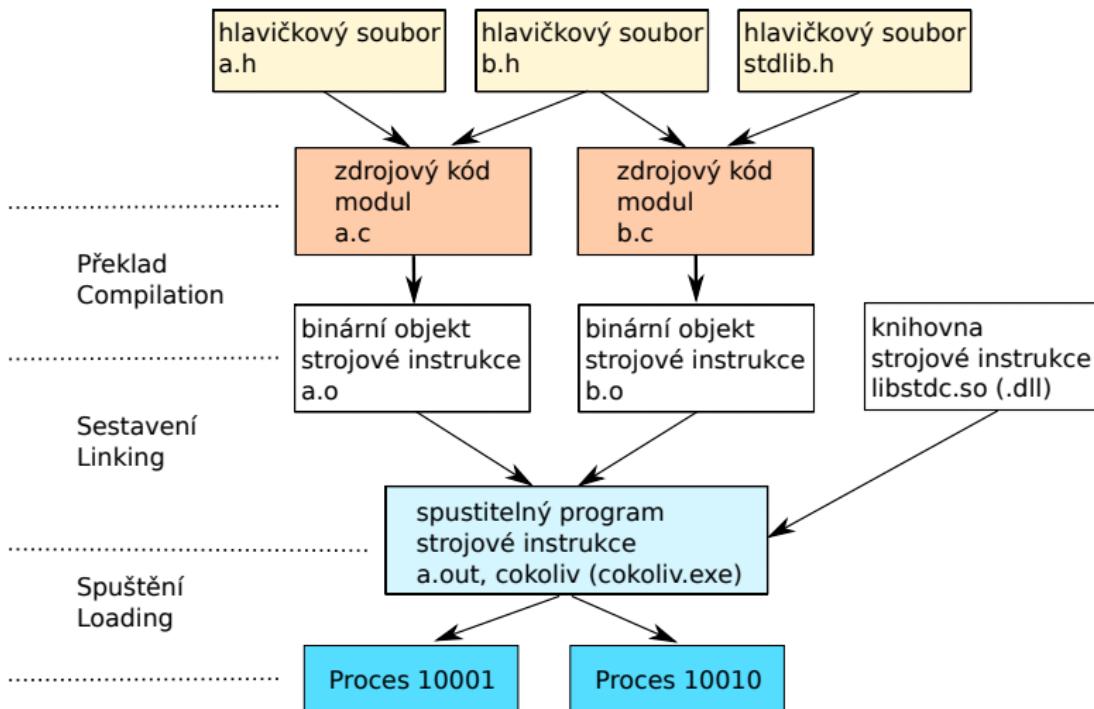
Nevýhody:

- nízká efektivita „běhu programu“
- interpret stále analyzuje zdrojový text (např. v cyklu) nebo se „simuluje“ jiný stroj

Poznámka:

- strojový kód je interpretován hardwarem – CPU

# Překlad



# Překladač

## Úkoly překladače (kompilátoru)

- kontrolovat správnost zdrojového kódu
- „porozumět“ zdrojovému textu programu a převést ho do vhodného „meziproduktu“, který lze dále zpracovávat bez jednoznačné souvislosti se zdrojovým jazykem
- základní výstup kompilátoru bude záviset na jeho typu
  - tzv. nativní překladač generuje kód stroje, na kterém sám pracuje
  - křížový překladač (cross-compiler) vytváří kód pro jinou hardwarovou platformu (např. na PC vyvíjíme program pro vestavěný mikropočítač s procesorem úplně jiné architektury, než má naše PC)
- mnohdy umí překladač generovat i ekvivalentní program v jazyku symbolických adres (assembler)
- častou funkcí překladače je i optimalizace kódu
  - např. dvě po sobě jdoucí čtení téže paměťové lokace jsou zbytečná
  - jde často o velmi pokročilé techniky závislé na cílové architektuře, na zdrojovém jazyku
  - optimalizace je časově náročná, a proto lze úrovně optimalizace volit jako parametr překladu
  - při vývoji algoritmu chceme rychlý překlad, při konečném překladu provozní verze programu žádáme rychlosť a efektivitu

# Struktura překladače jazyka C

- Předzpracování – preprocesing, vložení souborů a nahrazení maker (`#define`), podmíněný překlad, odstranění komentářů
  - výsledkem je upravený jeden textový soubor pro překlad
- Lexikální analýza
  - výsledek jsou tokeny – rozpoznání stavebních prvků
- Syntaktická a sémantická analýza
  - výsledkem je strom odvození a tabulka symbolů
- Generátor mezikódu
  - výsledkem je abstraktní strojový jazyk – three address code, java – soubory class
- Optimalizace – odstranění redundantních operací, optimalizace cyklů, atp.
  - výsledek optimalizovaný abstraktní kód
- Generátor kódu – přiřazení proměnných registrům
  - výsledkem je binární objekt, který obsahuje strojové instrukce a inicializovaná data

# Lexikální analýza

- Lexikální analýza
- převádí textové řetězce na sérii tokenů (též lexemů), tedy textových elementů detekovaného typu
- např. příkaz: sedm = 3 + 4 generuje tokeny
  - (sedm, IDENT), (=, ASSIGN\_OP), (3, NUM), (+, OPERATOR), (4, NUM)
- Již na této úrovni lze detektovat chyby typu „nelegální identifikátor“ (např. 1q)
- Tvorbu lexikálních analyzátorů lze mechanizovat pomocí programů typu lex nebo flex

# Syntaktická a sémantická analýza

- základem je bezkontextová gramaticka
- bezkontextová gramatika je speciálním případem formální gramatiky
- bezkontextová gramatika je čtveřice  $G = (V_N, V_T, P, S)$ , kde
  - $V_N$  je množina neterminálních symbolů, tj. symbolů které se nevyskytují v popisovaném jazyce
  - $V_T$  je množina terminálních symbolů, tj. symbolů ze kterých je tvořen jazyk, v našem případě jsou terminální symboly výsledkem lexikání analýzy
  - $P$  je množina přepisovacích pravidel, pro bezkontextovou gramatiku jsou pouze pravidla  $A \rightarrow \beta$ , kde  $A \in V_N$  a  $\beta \in (V_N \cup V_T)^*$ , tzn.  $\beta$  je libovolné slovo složené z terminálů i neterminálů
  - $S$  je počáteční symbol z množiny  $V_N$

# Syntaktická a sémantická analýza

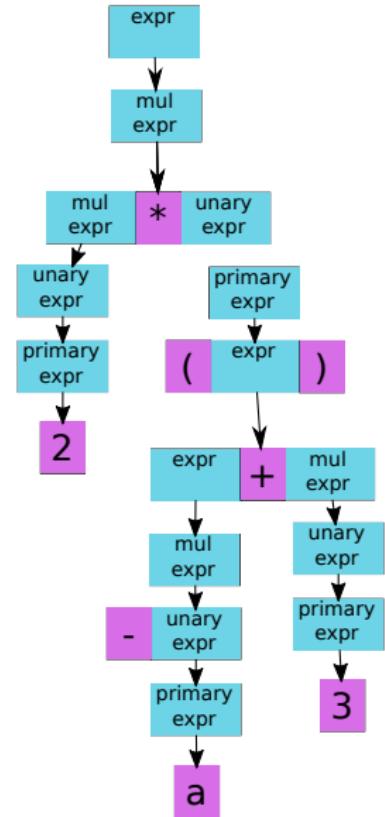
Příklad bezkontextová gramatika pro výrazy:

```

expr      : mul_expr
          | expr '+' mul_expr
          | expr '-' mul_expr;
mul_expr  : unary_expr
          | mul_expr '*' unary_expr
          | mul_expr '/' unary_expr;
unary_expr: primary_expr
          | unary_op primary_expr;
unary_op   : '&' | '*' | '+' | '-' | '~' | '!';
primary_expr: id | constant | '(' expr ')';
  
```

Odvození je pak již vlastně sémantická analýza (definuje význam věty - programu)

Vpravo vidíte strom odvození výrazu  $2*(-a+3)$



# Syntaktická a sémantická analýza

- většinou bývá prováděna společným kódem překladače, zvaným parser
- Tvorba parserů se mechanizuje pomocí programů typu yacc či bison
- yacc = Yet Another Compiler Compiler; bison je zvíře vypadající jako yacc
- Programovací jazyky se formálně popisují nejčastěji gramatikami pomocí Extended Backus-Naurovy Formy (EBNF)

```
digit_excluding_zero = "1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".  
digit              = "0" | digit_excluding_zero.  
natural_number      = digit_excluding_zero,{digit}.  
integer             = "0" | ["-"], natural_number.  
arit_operator       = "+" | "-" | "*" | "/".  
simple_int_expr     = integer,arit_operator,integer.
```

- EBNF pro jazyk C lze nalézt na  
[http://www.cs.man.ac.uk/~pjy/bnf/c\\_syntax.bnf](http://www.cs.man.ac.uk/~pjy/bnf/c_syntax.bnf)
- EBNF pro jazyk Java  
<http://cui.unige.ch/isi/bnf/JAVA/AJAVA.html>

# Three address code

- Celý program se popíše trojicemi: operand1 operace operand2
  - Některé operace mají jen jeden operand, druhý je nevyužit, např goto adresa.
- Každá trojice má svoje číslo, které obsahuje výsledek operace
- Nejčastější zápis: **t1:=op1 + op2**

Příklad: `for (i=0; i<10; i++) a+=i`

Příklad:  $x = \sqrt{a * a - b * b}$

`t1 := a * a`

`t2 := b * b`

`t3 := t1 - t2`

`t4 := push_param(t3)`

`t5 := call sqrt`

`t6 := x = t5`

`t1 := i = 0`

`t2 := goto t7`

`t3 := a + i`

`t4 := a = t3`

`t5 := i + 1`

`t6 := i = t5`

`t7 := i < 10`

`t8 := if t7 goto t3`

# Optimalizace při překladu

## Co může překladač optimalizovat

- Elementární optimalizace
  - předpočítání konstant
    - $n = 1024 * 64$  – během překladu se vytvoří konstanta 65536
  - znovupoužití vypočtených hodnot
    - `if(x**2 + y**2 <= 1) { a = x**2 + y**2; } else { a=0; }`
  - detekce a vyloučení nepoužitého kódu
    - `if((a>=0) && (a<0)) { never used code; };`
    - obvykle se generuje „upozornění“ (warning)
- Sémantické optimalizace
  - značně komplikované
  - optimalizace cyklů
  - lepší využití principu lokality (bude vysvětleno v části virtuální paměti)
  - minimalizace skoků v programu – lepší využití instrukční cache
- Celkově mohou být optimalizace velmi náročné během překladu, avšak za běhu programu mimořádně účinné (např. automatická paralelizace)

# Generování kódu

- Generátor kódu vytváří vlastní sémantiku "mezikódu"
  - Obecně: Syntaktický a sémantický analyzátor buduje strukturu programu ze zdrojového kódu, zatímco generátor kódu využívá tuto strukturální informaci (např. datové typy) k tvorbě výstupního kódu.
  - Generátor kódu mnohdy dále optimalizuje, zejména při znalosti cílové platformy
    - např.: Má-li cílový procesor více střádačů (datových registrů), dále nepoužívané mezivýsledky se uchovávají v nich a neukládají se do paměti.
- Podle typu překladu generuje různé výstupy
  - assembler (jazyk symbolických adres)
  - absolutní strojový kód
    - pro „jednoduché“ systémy (firmware vestavných systémů)
  - přemístitelný (object) modul
  - speciální kód pro pozdější interpretaci virtuálním strojem
    - např. Java byte-kód pro JVM
- V interpretačních systémech je generátor kódu nahrazen vlastním „interpretem“

# Binární objektový modul

Každý objektový modul obsahuje sérii sekcí různých typů a vlastností

- Prakticky všechny formáty objektových modulů obsahují
  - Sekce text obsahuje strojové instrukce a její vlastnosti je zpravidla EXEC|ALLOC
  - Sekce data slouží k alokaci paměťového prostoru pro inicializovaných proměnných, RW|ALLOC
  - Sekce BSS (Block Started by Symbol) popisuje místo v paměti, které netřeba alokovat ve spustitelném souboru, při start procesu je inicializováno na nulu, RW
- Mnohé formáty objektových modulů obsahují navíc
  - Sekce rodata slouží k alokaci paměťového prostoru konstant, RO|ALLOC
  - Sekci symtab obsahující tabulku globálních symbolů, kterou používá sestavovací program
  - Sekci dynamic obsahující informace pro dynamické sestavení
  - Sekci dynstr obsahující znakové řetězce (jména symbolů pro dynamické sestavení)
  - Sekci dynsym obsahující popisy globálních symbolů pro dynamické sestavení
  - Sekci debug obsahující informace pro symbolický ladící program
  - Detaily viz např.  
<http://www.freebsd.org/cgi/man.cgi?query=elf&sektion=5>

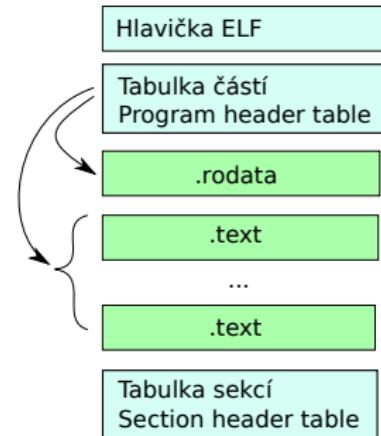
# Formáty binárního objektového modulu

- Různé operační systémy používají různé formáty jak objektových modulů tak i spustitelných souborů
- Existuje mnoho různých obecně užívaných konvencí
  - .com, .exe, a .obj
    - formát spustitelných souborů a objektových modulů v MSDOS
  - ELF – Executable and Linkable Format
    - nejpoužívanější formát spustitelných souborů, objektových modulů a dynamických knihoven v moderních implementacích POSIX systémů (Linux, Solaris, FreeBSD, NetBSD, OpenBSD, ...). Je též užíván např. i v PlayStation 2, PlayStation 3 a Symbian OS v9 mobilních telefonů.
    - Velmi obecný formát s podporou mnoha platform a způsobu práce s virtuální pamětí, včetně volitelné podpory ladění za běhu
  - Portable Executable (PE)
    - formát spustitelných souborů, objektových modulů a dynamických knihoven (.dll) ve MS-Windows. Označení "portable" poukazuje na univerzalitu formátu pro různé HW platformy, na nichž Windows běží.
  - COFF – Common Object File Format
    - formát spustitelných souborů, objektových modulů a dynamických knihoven v systémech na bázi UNIX V
    - Jako první zavedl sekce s explicitní podporou segmentace a virtuální paměti a obsahuje také sekce pro symbolické ladění

# Formát ELF

Formát ELF je shodný pro objektové moduly i pro spustitelné soubory

- ELF Header obsahuje celkové
  - popisné informace
  - např. identifikace cílového stroje a OS
  - typ souboru (obj vs. exec)
  - počet a velikostí sekci
  - odkaz na tabulku sekcí
  - ...
- Pro spustitelné soubory je podstatný seznam sekcí i modulů.
- Pro sestavování musí být moduly popsány svými sekciemi.
- Sekce jsou příslušných typů a obsahují „strojový kód“ či data
- Tabulka sekcí popisuje jejich typ, alokační a přístupové informace a další údaje potřebné pro práci sestavovacího či zaváděcího programu



# ELF struktura

Soubor NOVA kern/include/elf.h

```
class Eh {
public:
    uint32 ei_magic;
    uint8 ei_class, ei_data,
          ei_version, ei_pad[9];
    uint16 type, machine;
    uint32 version;
    mword entry, ph_offset,
          sh_offset;
    uint32 flags;
    uint16 eh_size, ph_size, ph_count,
          sh_size, sh_count, strtab;
};

enum {
    PF_X = 0x1,
    PF_W = 0x2,
    PF_R = 0x4,
};

```

```
class Ph {
public:
    enum {
        PT_NULL      = 0,
        PT_LOAD      = 1,
        PT_DYNAMIC   = 2,
        PT_INTERP    = 3,
        PT_NOTE      = 4,
        PT_SHLIB     = 5,
        PT_PHDR      = 6,
    };
    uint32 type;
    uint32 f_offs;
    uint32 v_addr;
    uint32 p_addr;
    uint32 f_size;
    uint32 m_size;
    uint32 flags;
    uint32 align;
};

```

# ELF struktura použití

```
int f = open(argv[1], O_RDONLY), i;
unsigned char buf[2048];
if (f>=0) {
    read(f, buf, sizeof(Eh));
    Eh *eh=(Eh *)buf;
    printf("Magic %08x - %c%c%c%c\n", eh->ei_magic, eh->ei_magic&0xff,
           (eh->ei_magic>>8)&0xff, (eh->ei_magic>>16)&0xff, (eh->ei_magic>>24)&0xff);
    printf("Entry point %08lx\n", eh->entry);
    printf("Program headers Num=%i, offset=%lu size Eh %i\n", eh->ph_count,
           eh->ph_offset, sizeof(Ph));

    int num = eh->ph_count;
    int read_num = eh->ph_offset+num*32;
    read(f, buf+sizeof(Eh), read_num-sizeof(Eh));

    for (i=0; i<num; i++) {
        Ph *ph=(Ph *)&buf[eh->ph_offset+i*32];
        printf("Program header n%i: type %i f_offs %08x, v_addr %08x, f_size %04x,
               flags %0x, align %i\n",
               i, ph->type & 0xff, ph->f_offs, ph->v_addr, ph->f_size, ph->flags, ph->align);
    }
}
```

## Sestavování a externí symboly

- V objektovém modulu jsou (asoň z hlediska sestavování) potlačeny lokální symboly (např. lokální proměnné uvnitř funkcí – jsou nahrazeny svými adresami, symbolický tvar má smysl jen pro případné ladění - debugging)
- globální symboly slouží pro vazby mezi moduly a jsou 2 typů
  - exportované symboly – jsou v příslušném modulu plně definovány, je známo jejich jméno a je známa i sekce, v níž se symbol vyskytuje a relativní adresa symbolu vůči počátku sekce.
  - importované symboly – symboly z cizích modulů, o kterých je známo jen jejich jméno, případně typ sekce, v níž by se symbol měl nacházet (např. pro odlišení, zda symbol představuje jméno funkce či jméno proměnné)

# Sestavování a externí symboly

Příklad z materiálů k přednáškám.

Soubor a.h:

```
extern int i_a;

int f_a(short int x);
```

Soubor b.c:

```
#include "a.h"

int i_b;
int f_b(int x) {
    i_b = f_a((short)x);
    i_a = (x/2);
    return (x>>16);
}
```

Tabulka symbolů (zkráceno)

```
Symbol table '.symtab' contains 12 entries:
Num: Value  Size Type  Bind  Vis   Ndx Name
 8: 000004    4 OBJECT GLOBAL DEFAULT COM i_b
 9: 000000  52 FUNC   GLOBAL DEFAULT    1 f_b
10: 000000     0 NOTYPE GLOBAL DEFAULT UND f_a
11: 000000     0 NOTYPE GLOBAL DEFAULT UND i_a
```

Sekce modulu b.o (readelf -a b.o)

Section Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[ 0]		NULL	0000	0000	0000	00		0	0	0
[ 1]	.text	PROGBITS	0000	0034	0034	00	AX	0	0	1
[ 2]	.rel.text	REL	0000	01a4	0018	08	I	9	1	4
[ 3]	.data	PROGBITS	0000	0068	0000	00	WA	0	0	1
[ 4]	.bss	NOBITS	0000	0068	0000	00	WA	0	0	1
[ 5]	.comment	PROGBITS	0000	0068	002a	01	MS	0	0	1
[ 6]	.note.GNU-stack		0000	0092	0000	00		0	0	1
[ 7]	.eh_frame	PROGBITS	0000	0094	0038	00	A	0	0	4
[ 8]	.rel.eh_frame									
		REL	0000	01bc	0008	08	I	9	7	4
[ 9]	.symtab	SYMTAB	0000	00cc	00c0	10		10	8	4
[10]	.strtab	STRTAB	0000	018c	0015	00		0	0	1
[11]	.shstrtab	STRTAB	0000	01c4	0057	00		0	0	1

# Sestavování a externí symboly

## Modul b.o

### Překlad funkce f\_b

00000000 <f\_b>:

				Offset	Info	Type	Sym. Value	Sym.	Name
0: 55		push	%ebp						
1: 89 e5		mov	%esp,%ebp						
3: 83 ec 08		sub	\$0x8,%esp						
6: 8b 45 08		mov	0x8(%ebp),%eax						
9: 98		cwtl							
a: 83 ec 0c		sub	\$0xc,%esp	0ffset	Info	Type	Sym. Value	Sym.	Name
d: 50		push	%eax	0000000f	00000a02	R_386_PC32	00000000	f_a	
e: e8 fc ff ff ff		call	f <f_b+0xf>	00000017	00000801	R_386_32	00000004	i_b	
13: 83 c4 10		add	\$0x10,%esp	00000028	00000b01	R_386_32	00000000	i_a	
16: a3 00 00 00 00		mov	%eax,0x0						
1b: 8b 45 08		mov	0x8(%ebp),%eax						
1e: 89 c2		mov	%eax,%edx						
20: c1 ea 1f		shr	\$0x1f,%edx						
23: 01 d0		add	%edx,%eax						
25: d1 f8		sar	%eax						
27: a3 00 00 00 00		mov	%eax,0x0						
2c: 8b 45 08		mov	0x8(%ebp),%eax						
2f: c1 f8 10		sar	\$0x10,%eax						
32: c9		leave							
33: c3		ret							

- Relokační tabulka musí obsahovat odkazy na symboly jejichž poloha není známá v době překladu (f\_a, i\_a)
- Relokační tabulka potřebuje i známý symbol i\_b, protože ve výsledném programu může být na jiném místě

# Sestavování a externí symboly

Funkce f\_b uvnitř výsledného programu

Před sestavením

00000000 <f\_b>:

```

0: 55          push    %ebp
1: 89 e5       mov      %esp,%ebp
3: 83 ec 08   sub     $0x8,%esp
6: 8b 45 08   mov     0x8(%ebp),%eax
9: 98          cwtl
a: 83 ec 0c   sub     $0xc,%esp
d: 50          push    %eax
e: e8 fc ff ff ff call   f <f_b+0xf>
13: 83 c4 10  add     $0x10,%esp
16: a3 00 00 00 00 mov    %eax,0x0
1b: 8b 45 08  mov     0x8(%ebp),%eax
1e: 89 c2       mov      %eax,%edx
20: c1 ea 1f   shr     $0x1f,%edx
23: 01 d0       add     %edx,%eax
25: d1 f8       sar     %eax
27: a3 00 00 00 00 mov    %eax,0x0
2c: 8b 45 08   mov     0x8(%ebp),%eax
2f: c1 f8 10   sar     $0x10,%eax
32: c9          leave
33: c3          ret

```

Po vytvoření programu

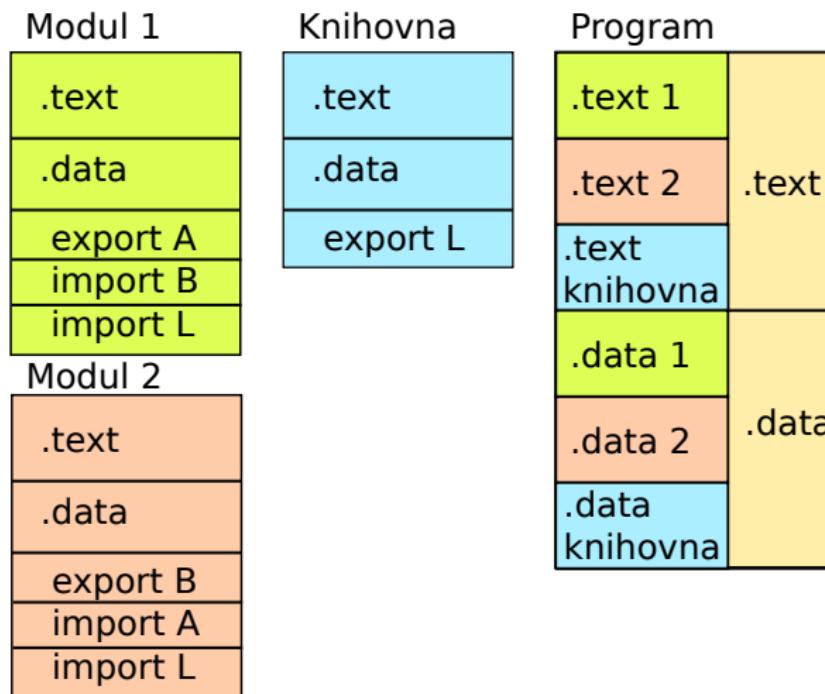
0000057f <f\_b>:

```

57f: 55          push    %ebp
580: 89 e5       mov      %esp,%ebp
582: 83 ec 08   sub     $0x8,%esp
585: 8b 45 08   mov     0x8(%ebp),%eax
588: 98          cwtl
589: 83 ec 0c   sub     $0xc,%esp
58c: 50          push    %eax
58d: e8 8b ff ff ff call   51d <f_a>
592: 83 c4 10  add     $0x10,%esp
595: a3 10 20 00 00 mov    %eax,0x2010
59a: 8b 45 08  mov     0x8(%ebp),%eax
59d: 89 c2       mov      %eax,%edx
59f: c1 ea 1f   shr     $0x1f,%edx
5a2: 01 d0       add     %edx,%eax
5a4: d1 f8       sar     %eax
5a6: a3 0c 20 00 00 mov    %eax,0x200c
5ab: 8b 45 08   mov     0x8(%ebp),%eax
5ae: c1 f8 10   sar     $0x10,%eax
5b1: c9          leave
5b2: c3          ret

```

# Externí symboly



## Statické knihovny

- Knihovna je vlastně balík binárních objektových modulů
- Podle symbolů požadovaných moduly programu se hledají moduly v knihovnách, které tyto symboly exportují
- Nový modul z knihovny může vyžadovat další symboly z dalších modulů
- Pokud po projití všech modulů programu i všech modulů knihovny není symbol nalezen, je ohlášena chyba a nelze sestavit výsledný program

# Dynamické knihovny

- Sestavovací program pracuje podobně jako při sestavování statickém, ale dynamické knihovny nepřidává do výsledného programu.
- Odkazy na symboly z dynamických knihoven je nutné vyřešit až při běhu programu. Existují v zásadě dva přístupy:
  - Vyřešení odkazů **při zavádění programu** do paměti – všechny nepropojené symboly extern se propojí před spuštěním programu
  - **Opožděné sestavování** – na místě nevyřešených odkazů připojí sestavovací program malé kousky kódu (zvané stub), které zavolají systém, aby odkaz vyřešil. Při běhu pak „stub“ zavolá operační systém, který zkontroluje, zda potřebná dynamická knihovna je v paměti (není-li zavede ji do paměti počítače), ve virtuální paměti knihovnu připojí tak, aby ji proces viděl. Následně stub nahradí správným odkazem do paměti a tento odkaz provede.
    - Výhodné z hlediska využití paměti, neboť se nezavádí knihovny, které nebudou potřeba.

# Dynamické knihovny PLT/GOT

V předcházejícím případě jsme použili dynamickou knihovnu glibc.

Začátek programu (zkráceno)

```
080482f0 <_start>:  
 80482f0: 31 ed           xor    %ebp,%ebp  
 ...  
 8048307: 68 03 84 04 08   push   $0x8048403  
 804830c: e8 cf ff ff ff  call   80482e0 <__libc_start_main@plt>
```

Funkce \_\_libc\_start\_main@plt je zodpovědná za dynamickou funkci start\_main z knihovny libc

```
080482e0 <__libc_start_main@plt>:  
 80482e0: ff 25 10 a0 04 08   jmp    *0x804a010  
 80482e6: 68 08 00 00 00       push   $0x8  
 80482eb: e9 d0 ff ff ff     jmp    80482c0 <_init+0x2c>
```

V okamžiku komplikace a spuštění je v paměti na adresě 0x804a010 hodnota 0x80482e6

Disassembly of section .got.plt:  
0804a000 <\_GLOBAL\_OFFSET\_TABLE\_>:

```
...  
804a010: e6 82 04 08
```

- Po zavedení knihovna dojde k přepsání GOT položky pro tuto funkci na správnou adresu funkce v paměti
- Při dalším volání tedy již instrukce jmp \*0x804a010 skočí přímo do funkce \_\_libc\_start\_main

# PIC a DLL

Dynamické knihovny jsou sdíleny různými procesy. Bud' musí být na stejné pozici/relokovaná (dll) nebo musí být na pozici nezávislé (PIC)

- PIC = Position Independent Code

- Překladač generuje kód nezávislý na umístění v paměti
- Skoky v kódu a odkazy na data jsou buď relativní vůči IP, nebo podle GOT – Global offset table
- Pokud nelze k adresaci použít registr IP, je nutné zjistit svoji polohu v paměti:

```
call .tmp1
.tmp1: pop %edi
        addl $_GLOBAL_OFFSET_TABLE - .tmp1, %edi
```

- pro x86\_64 lze použít pro adresaci registr RIP (číslo 0x2009db je posunutí GOT od prováděné instrukce, spočítáno překladačem)

```
mov $0x2009db(%rip), %rax
```

- kód je sice obvykle delší, avšak netřeba cokoliv modifikovat při sestavování či zavádění
- užívá se zejména pro dynamické knihovny
- v poslední době se využívá i pro programy

# DLL

- DLL – knihovna je na stejném místě pro všechny procesy
  - pokud se zavádí nová knihovna pro další process, pak musí být na volném místě
  - pokud není volné místo tam, kam je připravena, musí se relokovat – posunout všechny vnitřní pevné odkazy (skoky a data)
  - MS přiděluje místa v paměti na požadání vývojářů, aby minimalizoval možnost kolize
  - Vaše knihovna bude pravděpodobně v kolizi a bude se proto přesouvat – pomalejší provedení programu s touto knihovnou

# Zavaděč

- Zavaděč – loader – je zodpovědný za spuštění programu
- V POSIX systémech je to vlastně obsluha služby „execve“
- Úkoly zavaděče
  - vytvoření „obrazu procesu“ (memory image) v odkládacím prostoru na disku a částečně i v hlavní paměti v závislosti na strategii virtualizace, případné vyřešení nedefinovaných odkazů
  - sekce ze spustitelného souboru se stávají segmenty procesu (pokud správa paměti nepodporuje segmentaci, pak stránkami)
  - segmenty získávají příslušná „práva“ (RW, RO, EXEC, ...)
  - inicializace „registrování procesu“ v PCB
    - např. ukazatel zásobníku a čítač instrukcí
  - předání řízení na vstupní adresu procesu

# B4B350SY: Operační systémy

## Lekce 4. Plánování a synchronizace

Petr Štěpán

[stepan@fel.cvut.cz](mailto:stepan@fel.cvut.cz)



January 28, 2021

# Outline

1 Plánování procesů/vláken

2 Synchronizace

# Druhy plánovačů

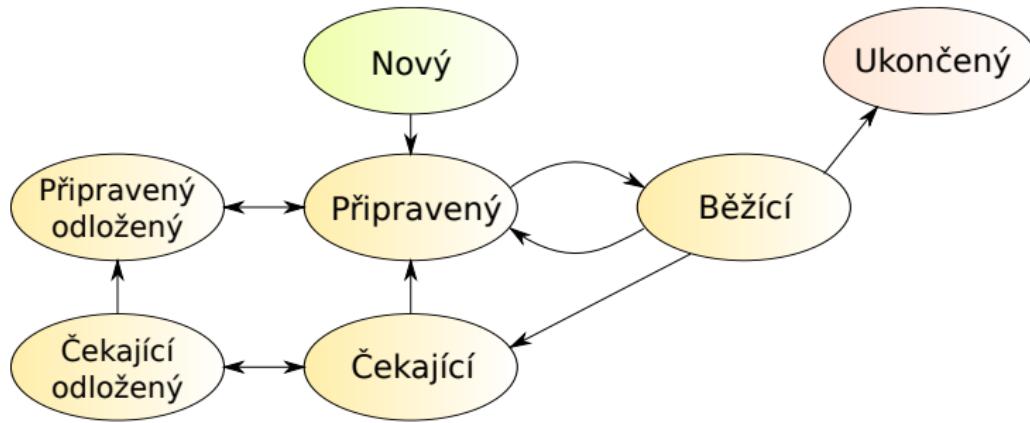
- Krátkodobý plánovač (operační plánovač, dispečer):
  - Základní správa procesoru/ů
  - Vybírá proces, který poběží na uvolněném procesoru přiděluje procesu procesor (CPU)
  - vyvoláván velmi často, musí být extrémně rychlý
- Střednědobý plánovač (taktický plánovač)
  - Úzce spolupracuje se správou hlavní paměti
  - Taktika využívání omezené kapacity fyzické paměti při multitaskingu
  - Vybírá, který proces je možno zařadit mezi odložené procesy
  - uvolní tím prostor zabíraný procesem v fyzické paměti
  - Vybírá, kterému odloženému procesu lze znova přidělit prostor v paměti počítače
- Dlouhodobý plánovač (strategický plánovač, job scheduler)
  - Vybírá, který požadavek na výpočet lze zařadit mezi procesy, a definuje tak stupeň multiprogramování
  - Je volán zřídka (jednotky až desítky sekund), nemusí být rychlý
  - V interaktivních systémech se používá velmi omezeně, např. plánování aktualizací

# Stavy procesu

Nové stavy spojené s odkládáním procesu na disk při nedostatku fyzické paměti:

- Odložený připravený
- Odložený čekající

Moderní OS většinou neprovádí odkládání celých procesů, ale při nedostatku paměti pak hrozí thrashing (podrobněji probereme při stránkování).



# Dispečer

- Dispečer pracuje s procesy, které jsou v hlavní paměti a jsou schopné běhu, tj. připravené (ready)
- Existují 2 typy plánování
  - nepreemptivní plánování (kooperativní plánování, někdy také plánování bez předbíhání)
    - běžícímu procesu nelze „násilně“ odejmout CPU, proces se musí procesoru vzdát, nebo ho nabídnout
    - historické operační systémy, kdy nebyla od systému podpora preempce
    - nyní se používá zpravidla jen v „uzavřených systémech“, kde jsou předem známy všechny procesy a jejich vlastnosti. Navíc musí být naprogramovány tak, aby samy uvolňovaly procesor ve prospěch procesů ostatních
  - preemptivní plánování (plánování s předbíháním),
- procesu schopnému dalšího běhu může být procesor odňat i „bez jeho souhlasu“, tedy kdykoliv
- plánovač rozhoduje v okamžiku:
  - 1 kdy některý proces přechází ze stavu běžící do stavu čekající nebo končí
  - 2 kdy některý proces přechází ze stavu čekající do stavu připravený
  - 3 přijde vnější podnět od HW prostřednictvím přerušení, nejčastěji od časovače
- První případ se vyskytuje v obou typech plánování
- Další dva jsou použity pouze pro plánování preemptivní

# Kritéria plánování

## Kritéria plánování

- Uživatelsky orientovaná
  - čas odezvy
    - doba od vzniku požadavku do reakce na něj
  - doba obrátky
    - doba od vzniku procesu do jeho dokončení
  - konečná lhůta (deadline)
    - požadavek dodržení stanoveného času dokončení
  - předvídatelnost
    - Úloha by měla být dokončena za zhruba stejnou dobu bez ohledu na celkovou zátěž systému
    - Je-li systém vytížen, prodloužení odezvy by mělo být rovnoměrně rozděleno mezi procesy
- Systémově orientovaná
  - průchodnost
    - počet procesů dokončených za jednotku času
  - využití procesoru
    - relativní čas procesoru věnovaný aplikačním procesům
  - spravedlivost
    - každý proces by měl dostat svůj čas (ne „hladovění“ či „stárnutí“)
  - vyvažování zátěže systémových prostředků
    - systémové prostředky (periferie, hlavní paměť) by měly být zatěžovány v čase rovnoměrně

# Základní plánovače

Ukážeme plánování:

- FCFS (First-Come First-Served)
- SPN (SJF) (Shortest Process Next)
- SRT (Shortest Remaining Time)
- cyklické (Round-Robin)
- zpětnovazební (Feedback)

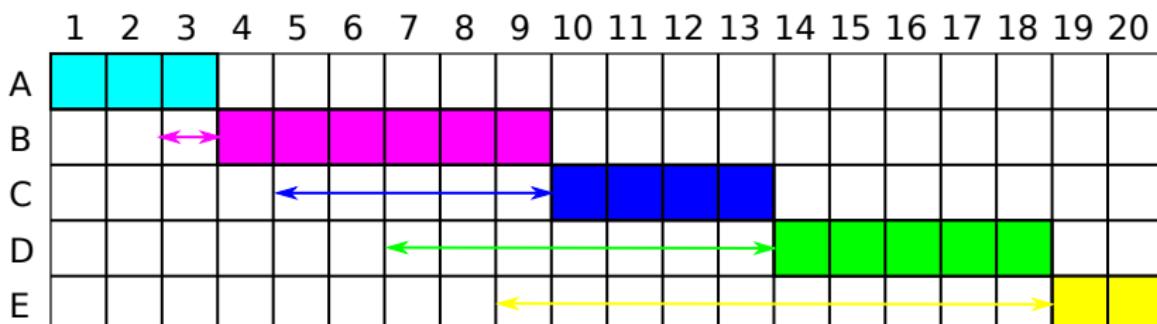
Příklad pro ilustraci algoritmů:

Proces	Čas příchodu	Potřebný čas
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

# FCFS

- FCFS = First Come First Served – prostá fronta FIFO
- Nejjednodušší nepreemptivní plánování
- Nově příchozí proces se zařadí na konec fronty
- Průměrné čekání může být velmi dlouhé

Příklad:



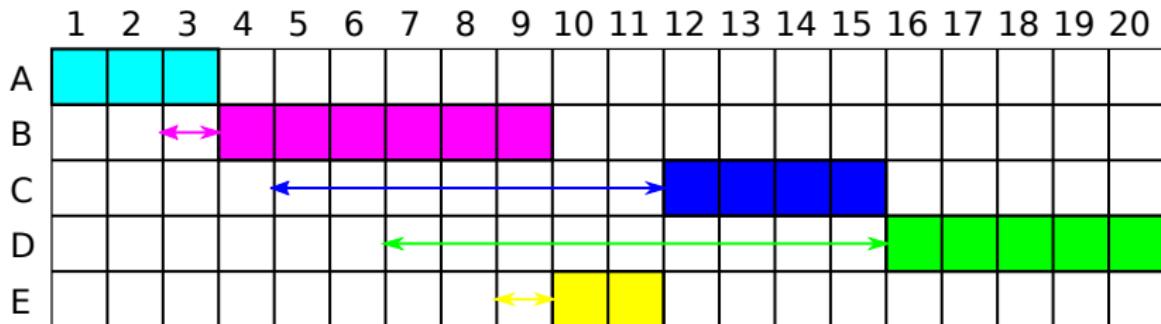
- Průměrné čekání  $T_{Avg} = \frac{0+1+5+7+10}{5} = 4.6$
- Průměrné čekání bychom mohli zredukovat, pokud by proces E běžel hned po B.

# FCFS – vlastnosti

- FCFS je primitivní nepreemptivní plánovací postup
- Průměrná doba čekání  $T_{Avg}$  silně závisí na pořadí přicházejících dávek
- Krátké procesy, které se připravily po dlouhém procesu, vytváří tzv. konvojový efekt
  - Všechny procesy čekají, až skončí dlouhý proces
- Pro krátkodobé plánování se FCFS samostatně fakticky nepoužívá.
  - Používá se pouze jako složka složitějších plánovacích postupů

# SPN

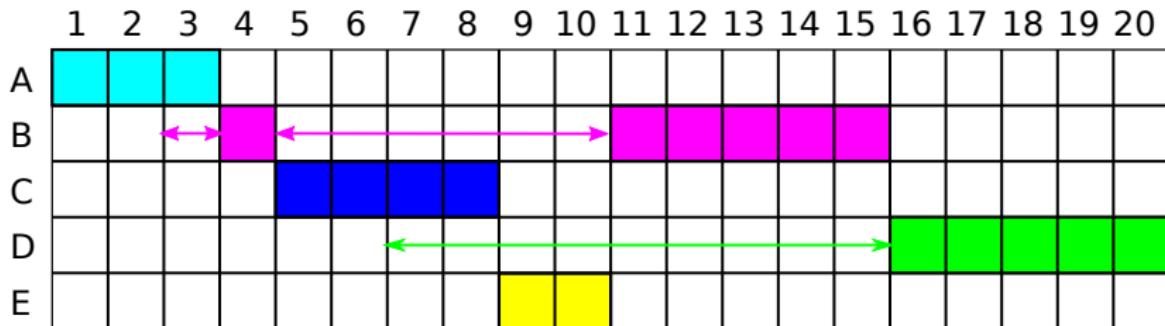
- SPN = Shortest Process Next (nejkratší proces jako příští); též nazýváno SJF = Shortest Job First
  - Opět nepreemptivní
  - Vybírá se připravený proces s nejkratší příští dávkou CPU
  - Krátké procesy předbíhají delší, nebezpečí stárnutí dlouhých procesů
  - Je-li kritériem kvality plánování průměrná doba čekání, je SPN optimálním algoritmem, což se dá exaktně dokázat



- Průměrné čekání  $T_{Avg} = \frac{0+1+7+9+1}{5} = 3.6$

# SRT

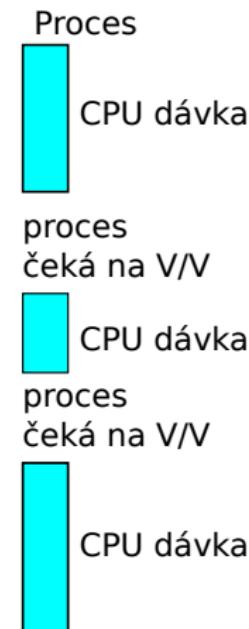
- SRT = Shortest Remaining Time (nejkratší zbývající čas)
- Preemptivní varianta SPN
- CPU dostane proces, který potřebuje nejméně času do svého ukončení
- Jestliže existuje proces, kterému zbývá k jeho dokončení čas kratší, než je čas zbývající do skončení procesu běžícího, dojde k preempcii
- Může existovat více procesů se stejným zbývajícím časem, a pak je nutno použít „arbitrážní pravidlo“, např. vybrat první z fronty



- Průměrné čekání  $T_{Avg} = \frac{0+7+0+9+0}{5} = 3.2$

# Jak nejlépe využít procesor

- Maximálního využití CPU se dosáhne uplatněním multiprogramování
- Jak ?
- Běh procesu = cykly alternujících dávek
  - CPU dávka
  - I/O dávka
- CPU dávka se může v čase překrývat s I/O dávkami dalších procesů



# Odhad délky běhu

- Délka příští dávky CPU skutečného procesu je známa jen ve velmi speciálních případech
  - Délka dávky se odhaduje na základě nedávné historie procesu
  - Nejčastěji se používá tzv. exponenciální průměrování
- Exponenciální průměrování
  - $t_n$  skutečná změřená délka n-té dávky CPU
  - $\tau_{n+1}$  odhad délky příští dávky CPU
  - $\alpha, 0 \leq \alpha \leq 1$  parametr vlivu historie
  - $\tau_{n+1} = \alpha \cdot t_n + (1-\alpha)\tau_n$
  - Příklad:
    - $\alpha = 0.5$
    - $\tau_{n+1} = 0.5 \cdot t_n + 0.5 \cdot \tau_n = 0.5 \cdot (t_n + \tau_n)$
    - $\tau_0$  se volí jako průměrná délka CPU dávky v systému nebo se odvodí z typu nejčastějších programů

# Prioritní plánování

- Každému procesu je přiřazeno prioritní číslo
  - Prioritní číslo – preference procesu při výběru procesu, kterému má být přiřazena CPU
  - CPU se přiděluje procesu s nejvyšší prioritou
  - Nejvyšší prioritě obvykle odpovídá (obvykle) nejnižší prioritní číslo
    - Ve Windows je to obráceně
- Existují opět dvě varianty:
  - Nepreemptivní
    - Jakmile se vybranému procesu procesor předá, procesor mu nebude odňat, dokud se jeho CPU dávka nedokončí
  - Preemptivní
    - Jakmile se ve frontě připravených objeví proces s prioritou vyšší, než je priorita právě běžícího procesu, nový proces předběhne právě běžící proces a odejmé mu procesor
- SPN i SRT jsou vlastně případy prioritního plánování
  - Prioritou je predikovaná délka příští CPU dávky
  - SPN je nepreemptivní prioritní plánování
  - SRT je preemptivní prioritní plánování

# Prioritní plánování – problémy

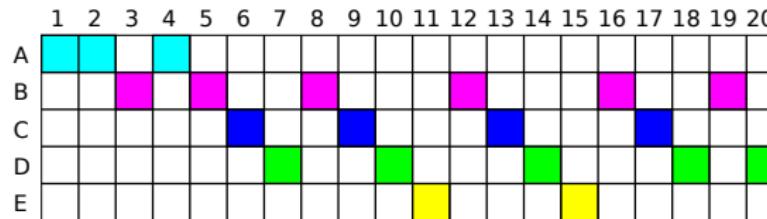
- Problém stárnutí (starvation):
  - Procesy s nízkou prioritou nikdy nepoběží; nikdy na ně nepřijde řada
    - Údajně: Když po řadě let vypínali v roce 1973 na M.I.T. svůj IBM 7094 (jeden z největších strojů své doby), našli proces s nízkou prioritou, který čekal od roku 1967.
- Řešení problému stárnutí: zrání procesů (aging)
  - Je nutno dovolit, aby se procesu zvyšovala priorita na základě jeho historie a doby setrvávání ve frontě připravených
    - Během čekání na procesor se priorita procesu zvyšuje

# Cyklické plánování

- Cyklická obsluha (Round-robin) – RR
- Z principu preemptivní plánování
- Každý proces dostává CPU periodicky na malý časový úsek, tzv. časové kvantum, délky  $q$  (desítky ms)
- V „čistém“ RR se uvažuje shodná priorita všech procesů
- Po vyčerpání kvanta je běžícímu procesu odňato CPU ve prospěch nejstaršího procesu ve frontě připravených a dosud běžící proces se zařazuje na konec této fronty
- Je-li ve frontě připravených procesů  $n$  procesů, pak každý proces získává  $\frac{1}{n}$  doby CPU
- Žádný proces nedostane 2 kvanta za sebou (samozřejmě pokud není jediný připravený)
- Žádný proces nečeká na začátek přidělení CPU déle než  $q(n-1)$

# Cyklické plánování

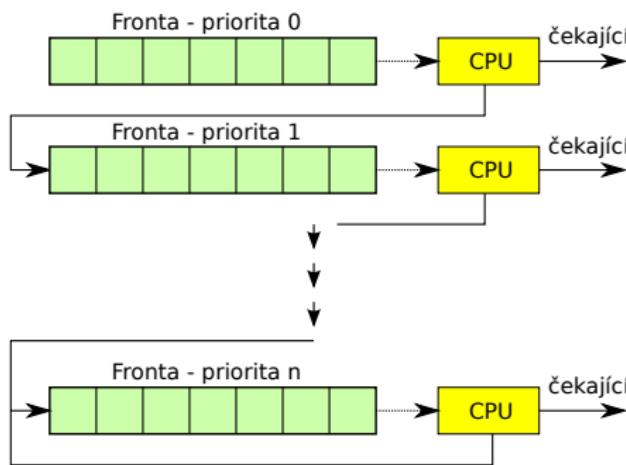
- Efektivita silně závisí na velikosti kvanta
- Velké kvantum – blíží se chování FCFS
  - Procesy dokončí svoji CPU dávku dříve, než jim vyprší kvantum.
- Malé kvantum – časté přepínání kontextu
  - značná režie
- Dosahuje se průměrné doby obrátky delší oproti plánování SRT
  - Průměrná doba obrátky se může zlepšit, pokud většina procesů se době **q** ukončí
  - Empirické pravidlo pro stanovení **q**: cca 80% procesů by nemělo vyčerpat kvantum
- Výrazně lepší je čas odezvy



# Zpětnovazební plánování

- Základní problém:
  - Neznáme předem časy, které budou procesy potřebovat
- Východisko:
  - Penalizace procesů, které běžely dlouho
- Řešení:
  - Dojde-li k preemci přečerpáním časového kvanta, procesu se snižuje priorita
  - Implementace pomocí víceúrovňových front
    - pro každou prioritu jedna
    - Nad každou frontou samostatně běží algoritmus určitého typu plánování, obvykle RR s různými kvanty a FCFS pro frontu s nejnižší prioritou

# Víceúrovňové zpětnovazební fronty



- Proces opouštějící procesor kvůli vyčerpání časového kvanta je přeřazen do fronty s nižší prioritou
- Fronty s nižší prioritou mohou mít delší kvanta
- Problém stárnutí ve frontě s nejnižší prioritou
  - Řeší se pomocí zrání (aging) – v jistých časových intervalech ( $\approx 10$  s) se zvyšuje procesům priorita přemístěním do „vyšších“ front

# O(1) plánovač – Linux 2.6.22

- O(1) – rychlosť plánovače nezávisí na počtu běžících procesů – je rychlý a deterministický
- Dvě sady víceúrovňových front
  - Na začátku první sada obsahuje připravené procesy, druhá je prázdná
  - Při vyčerpání časového kvanta je proces přeřazen do druhé sady front do nové úrovně
  - Vzbuzené procesy jsou zařazovány podle toho, zda ještě nevyužily celé svoje časové kvantum do aktivní sady front, nebo do druhé sady front
  - Pokud je první sada prázdná, dojde k prohození první a druhé sady front procesů
- Heuristika pro odhad interaktivních procesů a jejich udržování na nejvyšších prioritách s odpovídajícími časovými kvanty

# Zcela férový plánovač

- Linux od verze 2.6.23 (Completely Fair Scheduler)
- Nepoužívá fronty, ale jednu strukturu, která udržuje všechny procesy uspořádané podle délky již spotřebovaného času a délky čekání
  - kritérium = spotřebovaný\_čas - férový\_čekací\_čas
  - férový\_čekací\_čas je reálný čas dělený počtem čekajících procesů na jeden procesor
  - ideálně všechny procesy mají kritérium 0
- Pro rychlou implementaci se používá vyvážený binární červeno-černý strom, zaručující složitost úměrnou  $\log(n)$  počtu připravených procesů
- Nepotřebuje složité heuristiky pro detekci interaktivních procesů
- Jediný parametr je časové kvantum:
  - pro uživatelské PC se volí menší
  - pro serverové počítače větší kvanta omezují režii s přepínáním procesů a tím zvyšuje propustnost serveru
- Žádný proces nemůže zestárnout, všechny procesy mají stejné podmínky

# Plánování v multiprocesorech

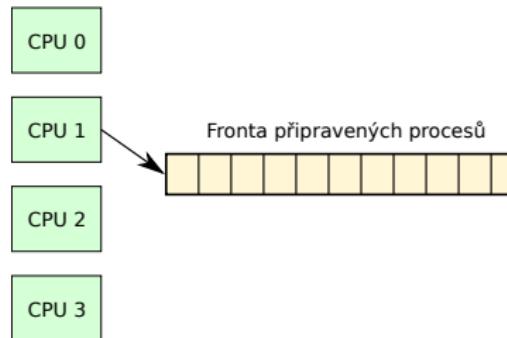
Přiřazování procesů (vláken) procesorům:

- Architektura „master/slave“
  - Klíčové funkce jádra běží vždy na jednom konkrétním procesoru
  - Master odpovídá za plánování
  - Slave žádá o služby mastera
  - Nevýhoda: dedikace
    - Přetížený master se stává úzkým místem systému
- Symetrický multiprocesing (SMP)
  - Všechny procesory jsou si navzájem rovny
  - Funkce jádra mohou běžet na kterémkoliv procesoru
  - SMP vyžaduje podporu vláken v jádře
  - Proces musí být dělen na vlákna, aby SMP byl účinný
- Aplikace je sada vláken pracujících paralelně do společného adresního prostoru
- Vlákno běží nezávisle na ostatních vláknech svého procesu
- Vlákna běžící na různých procesorech dramaticky zvyšují účinnost systému

# SMP

Dvě řešení SMP:

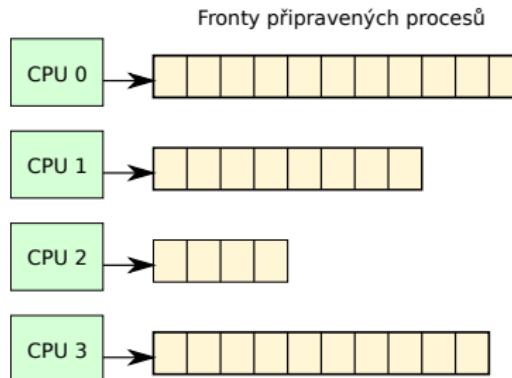
- Jedna společná fronta pro všechny procesory
  - Fronta může být víceúrovňová dle priorit
  - Problémy:
    - Jedna centrální fronta připravených sledů vyžaduje používání vzájemného vyloučování v jádře
    - Kritické místo v okamžiku, kdy si hledá práci více procesorů
    - Předběhnutá (přerušená) vlákna nebudou nutně pokračovat na stejném procesoru – nelze proto plně využívat cache paměti procesorů



# SMP

Druhé řešení SMP:

- Každý procesor má svojí frontu a občasná migrace vláken mezi procesory má za úkol udržovat fronty přibližně stejně dlouhé
  - Každý procesor si sám vyhledává příští vlákno
  - Přesněji: instance plánovače běžící na procesoru si je sama vyhledává
  - Problémy – některé fronty jsou kratší:
    - Heuristická pravidla, kdy frontu změnit



# SMP optimalizace

- Používají se různá (heuristická) pravidla (i při globální frontě):
  - Afinita vlákna k CPU – použij procesor, kde vlákno již běželo (možná, že v cache CPU budou ještě údaje z minulého běhu)
  - Afinita vlákna k CPU při globální frontě – neber první proces z fronty, ale prozkoumej více procesů na začátku fronty a hledej proces, který běžel na daném procesoru
  - Použij nejméně využívaný procesor
- Mnohdy značně složité
  - při malém počtu procesorů ( $\leq 4$ ) může přílišná snaha o optimalizaci plánování vést až k poklesu výkonu systému, výběr se dělá při každém rozhodování, kdo poběží
    - Tedy aspoň v tom smyslu, že výkon systému neporoste lineárně s počtem procesorů
  - při velkém počtu procesorů dojde naopak k „nasycení“, neboť plánovač se musí věnovat rozhodování velmi často (končí CPU dávky na mnoha procesorech)
    - režie tak neúměrně roste

# Jak to bude fungovat?

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

volatile int a;
void *fce(void *n) {
    int i;
    for (i=0; i<10000; i++) {
        a+=1;
    }
    printf("a=%i\n", a);
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    pthread_t tid1,tid2;
    a=0;
    pthread_create(&tid1, NULL, fce, NULL);
    pthread_create(&tid2, NULL, fce, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    return 0;
}
```

# Problém synchronizace

- Souběžný přístup ke sdíleným datům může způsobit jejich nekonzistenci
  - nutná koordinace procesů
- Synchronizace běhu procesů
  - Čekání na událost vyvolanou jiným procesem
- Komunikace mezi procesy (IPC = Inter-process Communication) – příští přednáška
  - Výměna informací (zpráv)
  - Způsob synchronizace, koordinace různých aktivit
- Sdílení prostředků – problém soupeření či souběhu (race condition)
  - Procesy používají a modifikují sdílená data
  - Operace zápisu musí být vzájemně výlučné
  - Operace zápisu musí být vzájemně výlučné s operacemi čtení
  - Operace čtení (bez modifikace) mohou být realizovány souběžně
  - Pro zabezpečení integrity dat se používají kritické sekce

# Producent konzument

## Ilustrační příklad

- Producent generuje data do vyrovnávací paměti s konečnou kapacitou (bounded-buffer problem) a konzument z této paměti data odebírá
- Zavedeme celočíselnou proměnnou count, která bude čítat platné položky v bufferu. Na počátku je  $\text{count} = 0$
- Pokud je v poli místo, producent vloží položku do pole a inkrementuje count
- Pokud je v poli nějaká položka, konzument při jejím vyjmutí dekrementuje count

# Producent a konzument

Sdílená data

```
#define BUF_SIZE = 20
typedef struct { /* data */ } item;
item buffer[BUF_SIZE];
int count = 0;
```

Producent

```
void producer() {
    int in = 0;
    item nextProduced;
    while (1) {
        /* Vytvoř novou položku do
           proměnné nextProduced */
        while (count == BUF_SIZE);
        /* čekání nedělej nic */
        buffer[in] = nextProduced;
        in = (in + 1) % BUF_SIZE;
        count++;
    }
}
```

Kde je problém?

Konzument

```
void consumer() {
    int out = 0;
    item nextConsumed;
    while (1) {
        while (count == 0);
        /* čekání nedělej nic */
        nextConsumed = buffer[out];
        out = (out + 1) % BUF_SIZE;
        count--;
        /* Zpracuj položku z
           proměnné nextConsumed */
    }
}
```

# Problém soupeření

- count ++ bude obvykle implementováno:

$P_1$ : **count** → registr      mov count, %eax  
 ■  $P_2$ : registr+1 → registr      add 1, %eax  
 $P_3$ : registr → **count**      mov %eax, count

- count -- bude obvykle implementováno:

$K_1$ : **count** → registr      mov count, %eax  
 ■  $K_2$ : registr-1 → registr      sub 1, %eax  
 $K_3$ : registr → **count**      mov %eax, count

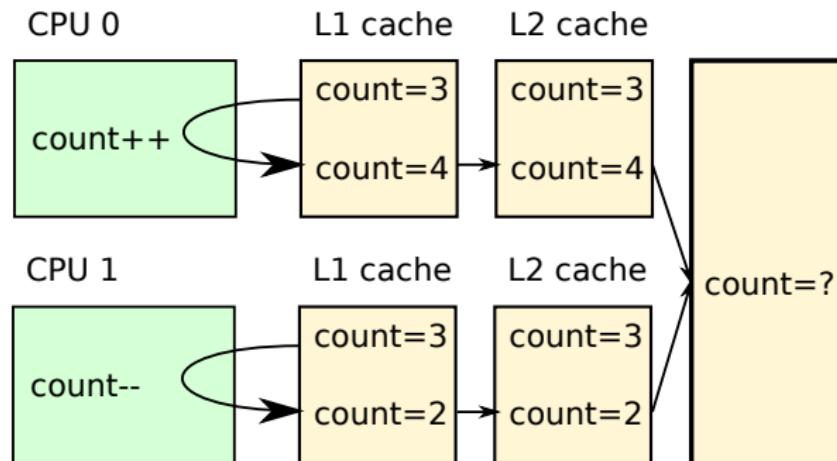
Může nastat následující paralelizace procesů konzument a producent:

akce	běží	akce	výsledek
$P_1$ :	producent	<b>count</b> → registr	eax = 3
$P_2$ :	producent	registr+1 → registr	eax = 4
$K_1$ :	konzument	<b>count</b> → registr	eax = 3
$K_2$ :	konzument	registr-1 → registr	eax = 2
$K_3$ :	konzument	registr → <b>count</b>	count = 2
$P_3$ :	producent	registr → <b>count</b>	count = 4

- Na konci může být count roven 2 nebo 4, ale správně je 3 (což se většinou podaří)
- Je to důsledkem nepředvídatelného prokládání procesů/vláken vlivem možné preempce

# Problém soupeření – cache

- Problém soupeření je i při vícejádrových procesorech
- Jedna proměnná je uložena na více místech cache úrovně L1, úrovně L2 a pouze jednom místě úrovně L3 a paměti RAM



- Výsledek zápisu je určen kdo přijde dříve a kdo později, ale pouze hodnota 2 nebo 4

# Kritická sekce

- Problém lze formulovat obecně:
  - Jistý čas se proces zabývá svými obvyklými činnostmi a jistou část své aktivity věnuje sdíleným prostředkům.
  - Část kódu programu, kde se přistupuje ke sdílenému prostředku, se nazývá kritická sekce procesu vzhledem k tomuto sdílenému prostředku (nebo také sdružená s tímto prostředkem).
- Je potřeba zajistit, aby v kritické sekci sdružené s jistým prostředkem, se nacházel nejvýše jeden proces
  - Pokud se nám podaří zajistit, aby žádné dva procesy nebyly současně ve svých kritických sekcích sdružených s uvažovaným sdíleným prostředkem, pak je problém soupeření vyřešen.
- Modelové prostředí pro řešení problému kritické sekce
  - Předpokládá se, že každý z procesů běží nenulovou rychlosťí
  - Řešení nesmí záviset na relativních rychlostech procesů

# Požadavky na kritickou sekci

- Vzájemné vyloučení – podmínka bezpečnosti (Mutual Exclusion)
  - Pokud proces  $P_i$  je ve své kritické sekci, pak žádný další proces nesmí být ve své kritické sekci sdružené s týmž prostředkem
- Trvalost postupu – podmínka živosti (Progress)
  - Jestliže žádný proces neprovádí svoji kritickou sekci sdruženou s jistým zdrojem a existuje alespoň jeden proces, který si přeje vstoupit do kritické sekce sdružené se tímto zdrojem, pak výběr procesu, který do takové kritické sekce vstoupí, se nesmí odkládat nekonečně dlouho.
- Konečné čekání – podmínka spravedlivosti (Fairness)
  - Proces smí čekat na povolení vstupu do kritické sekce jen konečnou dobu.
  - Musí existovat omezení počtu, kolikrát může být povolen vstup do kritické sekce sdružené se jistým prostředkem jiným procesům než procesu požadujícímu vstup v době mezi vydáním žádosti a jejím uspokojením.

# Řešení kritických sekcí

Základní struktura procesu s kritickou sekcí:

```
do {  
    enter_cs();  
    // critical section  
    leave_cs ();  
    // non-critical section  
} while (TRUE);
```

Klíčem k řešení celého problému kritických sekcí je korektní implementace funkcí enter\_cs() a leave\_cs().

- Čistě softwarová řešení na aplikační úrovni
  - Algoritmy, jejichž správnost se nespoléhá na další podporu
  - Základní (a problematické) řešení s aktivním čekáním (busy waiting)
- Hardwarové řešení
  - Pomocí speciálních instrukcí CPU
  - Stále ještě s aktivním čekáním
- Softwarové řešení zprostředkováno operačním systémem
  - Potřebné služby a datové struktury poskytuje OS (např. semafory)
  - Tím je umožněno pasivní čekání – proces nesoutěží o procesor
  - Podpora volání synchronizačních služeb v programovacích systémech/jazycích (např. monitory, zasílání zpráv)

# Řešení na aplikační úrovni

Zaveděme proměnnou lock, jejíž hodnota určuje, zda je kritická sekce obsazená

```
while(TRUE) {  
    while(lock!=0);  
    /* čekej */  
    lock = 1;  
    critical_section();  
    lock = 0;  
    noncritical_section();  
}
```

Je zde nějaký problém?

- **Je to úplně špatně!**
- Protože mezi otestováním proměnné lock a jejím nastavení je možné, že proběhne další otestování jiným vláknem.
- Neřeší tedy základní podmítku exkluzivity kritické sekce

# Řešení na aplikační úrovni

Striktní střídání dvou procesů nebo vláken.

- Zavedeme proměnnou `turn`, jejíž hodnota určuje, který z procesů smí vstoupit do kritické sekce.
- Je-li `turn == 0`, do kritické sekce může  $P_0$ ,
- je-li `turn == 1`, pak  $P_1$ .

$P_0$

```
while(TRUE) {
    while(turn!=0);
        /* čekej */
    critical_section();
    turn = 1;
    noncritical_section();
}
```

$P_1$

```
while(TRUE) {
    while(turn!=1);
        /* čekej */
    critical_section();
    turn = 0;
    noncritical_section();
}
```

Je zde nějaký problém?

- $P_0$  proběhne svojí kritickou sekcí velmi rychle,  $turn = 1$  a oba procesy jsou v nekritických částech.  $P_0$  je rychlý i ve své nekritické části a chce vstoupit do kritické sekce. Protože však `turn == 1`, bude čekat, přestože kritická sekce je volná.
- Je porušen požadavek Trvalosti postupu
- Navíc řešení nepřípustně závisí na rychlostech procesů

# Petersonovo řešení

- Petersonovo řešení střídání dvou procesů nebo vláken
- Řešení pro dva procesy  $P_i$  ( $i = 0, 1$ ) – dvě globální proměnné:
  - int turn;
  - Proměnná turn udává, který z procesů je na řadě při přístupu do kritické sekce
  - boolean interest[2] ;
  - V poli interest procesy indikují svůj zájem vstoupit do kritické sekce; ( $interest[i]==TRUE$ ) znamená, že  $P_i$  tuto potřebu má
  - Prvky pole interest nejsou sdílenými proměnnými.

```
j = 1 - i;
interest[i] = TRUE;
turn = j;
while (interest[j] && turn == j) ;           /* čekání */
/* KRITICKÁ SEKCE */
interest[i] = FALSE;
/* NEKRITICKÁ ČÁST PROCESU */
```

- Náš proces bude čekat jen pokud druhý proces je na řadě a současně má zájem do kritické sekce vstoupit
- Všechna řešení na aplikační úrovni obsahují aktivní čekání, nebo používají funkci sleep/usleep

# Petersonovo řešení

```
int a;
volatile int turn;
volatile int interest[2];

void *fce(void *n) {
    int i;
    int j=*(int*)n;
    for (i=0; i<1000000; i++) {
        interest[j]=1;
        __sync_synchronize();      /* memory barrier */
        turn = (1-j);
        while (interest[1-j]==1 && turn==(1-j)); /* repeat and wait */
        a+=1;
        interest[j]=0;
    }
    printf("a=%i\n", a);
    pthread_exit(NULL);
}
```

# Memory barrier

- Většina moderních CPU umí měnit pořadí dvou po sobě jdoucích instrukcí kvůli zrychlení přístupu do paměti.
- Pro Petersonovo řešení je pořadí zápisu do proměnných turn a interest klíčové
- `__sync_synchronize` memory barrier pro překladač gcc (visual studio má funkci MemoryBarrier)
- `memory barrier` umožní i synchronizaci cache pamětí

```
interest[i] = TRUE;  
__sync_synchronize(); /* memory barrier */  
turn = j;  
while (interest[j] && turn == j) ;           /* čekání */
```

- Nyní je všechno v pořádku a řešení funguje

# Petersonovo řešení

Obecné řešení pro N procesů

- je již daleko více komplikovanější, tím je náchylnější k implementační chybě
- proměnné level charakterizují, kdo čeká na kritickou sekci
- proces, který dospěje až do nejvyšší úrovně (level), tak získá kritickou sekci

```
int level[N]
int last_to_enter[N-1]
for (l=0; l<N-1; l++)
    level[i] = 1
    last_to_enter[l] = i
while (last_to_enter[l] == i and exists k != i; level[k] >= 1)
    wait;
```

- konstrukce exist k je zkratka za

```
set = False
for (k=0; k<N; k++) {
    if (k!=i && level[k] >= 1)
        set = True;
```

# HW podpora

- Využití zamykací proměnné je rozumné, avšak je nutná atomicita
- Jednoprocesorové systémy mohou vypnout přerušení, při vypnutém přerušení nemůže dojít k preempcii
  - Nelze použít na aplikační úrovni (vypnutí přerušení je privilegovaná akce)
  - Nelze jednoduše použít pro víceprocesorové systémy
- Moderní systémy nabízejí speciální nedělitelné (atomicke) instrukce
  - Tyto instrukce mezi paměťovými cykly „nepustí“ sběrnici pro jiný procesor
  - Instrukce TestAndSet atomicky přečte obsah adresované buňky a bezprostředně poté změní její obsah (tas – MC68k, tsl – Intel)
  - Instrukce Swap (xchg) atomicky prohodí obsah registru procesoru a adresované buňky
  - Např. IA32/64 (I586+) nabízí i další atomicke instrukce
  - Prefix „LOCK“ pro celou řadu instrukcí typu read-modify-write (např. ADD, AND, ... s cílovým operandem v paměti)

# HW podpora

## ■ tas např. Motorola 68000

```
enter_cs:  tas  lock      ; nastav lock na 1 a otestuj starou hodnotu
            jnz enter_cs    ; byla stará hodnota nenulová?
            ret
```

```
leave_cs:   mov $0, lock    ; vynuluj lock pro uvolnění kritické sekce
            ret
```

## ■ xchg – IA32

```
enter_cs:  mov $1, %eax    ; připrav hodnotu 1 pro výměnu
            xchg lock, %eax ; eax obsahuje nyní starou hodnotu
            jnz enter_cs     ; byla stará hodnota nenulová
            ret
```

```
leave_cs:   mov $0, lock    ; vynuluj lock pro uvolnění kritické sekce
            ret
```

# Xchg řešení

```
volatile int a;
volatile int turn;
void *fce(void *n) {
    int i, tmp;
    for (i=0; i<1000000; i++) {
        tmp=1;
        asm volatile ("xchg%z0 %2, %0;" : "=g" (turn), "=r" (tmp): "1" (tmp) : );
        while (tmp!=0) {
            asm volatile ("xchg%z0 %2, %0;" : "=g" (turn), "=r" (tmp): "1" (tmp) : );
        }
        a+=1;
        turn=0;
    }
    printf("a=%i\n", a);
    pthread_exit(NULL);
}
```

# Synchronizace bez aktivního čekání

- Aktivní čekání mrhá strojovým časem
- Může způsobit i nefunkčnost při rozdílných prioritách procesů
- Např. vysokoprioritní producent zaplní pole, začne aktivně čekat a nedovolí konzumentovi odebrat položku (samozřejmě to závisí na metodě plánování procesů a na to navazující dynamicky se měnící priority)
- Blokování pomocí systémových atomických primitiv
  - suspend() místo aktivního čekání – proces se zablokuje
  - wakeup(process) probuzení spolupracujícího procesu při opouštění kritické sekce

```

void producer() {
    while (1) {
        /* Vygeneruj položku do proměnné nextProduced */
        if (count == BUFFER_SIZE) suspend();           // Je-li pole plné, zablokuj se
        buffer[in] = nextProduced;   in = (in + 1) % BUFFER_SIZE;
        count++ ;
        if (count == 1) wakeup(consumer);           // Bylo-li pole prázdné, probud' konzumenta
    }
}

void consumer() {
    while (1) {
        if (count == 0) suspend();           // Je-li pole prázdné, zablokuj se
        nextConsumed = buffer[out];   out = (out + 1) % BUFFER_SIZE;
        count-- ;
        if (count == BUFFER_SIZE-1)           // Bylo-li pole plné, probud' producenta
            wakeup(producer);
        /* Zpracuj položku z proměnné nextConsumed */
    }
}

```

# Problém s čekáním

- Předešlý kód není řešením – zůstalo konkurenční soupeření – count je opět sdílenou proměnnou:
  - Konzument přečetl count == 0 a než zavolá suspend(), je mu odňat procesor
  - Producent vloží do pole položku a count == 1, načež se pokusí se probudit konzumenta, který ale ještě nespí!
  - Po znovuspustění se konzument domnívá, že pole je prázdné a volá suspend()
  - Po čase producent zaplní pole a rovněž zavolá suspend() – spí oba!
  - Příčinou této situace je ztráta budícího signálu
- Lepší řešení:
  - Jednинě OS umí uspat a vzbudit procesy – Semafora, mutexy

# Semafor

- Obecný synchronizační nástroj (Edsger Dijkstra, NL, [1930–2002])
- Systémem spravovaný objekt
- Základní vlastností je celočíselná proměnná (obecný semafor, nebo také čítající semafor)
- Dvě standardní atomické operace nad semaforem
  - `sem_wait(&S)` [někdy nazývaná `lock()`, `acquire()` nebo `down()`]
  - `sem_post(&S)` [někdy nazývaná `unlock()`, `release()` nebo `up()`]

```

sem_wait(S) {
    while (S <= 0);
    S--;
}
sem_post(S) {
    S++;
    // Čeká-li jiný proces před
    // semaforem, pust' ho dál
}
  
```

- Tato sémantika stále obsahuje aktivní čekání
- Skutečná implementace však aktivní čekání obchází tím, že spolupracuje s plánovačem CPU, což umožňuje blokovat a reaktivovat procesy (vlákna)

# Implementace semaforů

Struktura semaforu

```
typedef struct {
    int value;           // „Hodnota“ semaforu
    struct process *list; // Fronta procesů stojících „před semaforem“
} sem_t;
```

Operace nad semaforem jsou pak implementovány jako nedělitelné s touto sémantikou

```
void sem_wait(sem_t *S) {
    S->value = S->value - 1;
    if (S->value < 0)           // Je-li třeba, zablokuj volající proces a zařaď ho
        block(S->list);       // do fronty před semaforem (S.list)
}

void sem_post(sem_t *S) {
    S->value = S->value + 1
    if (S->value <= 0) {
        if (S->list != NULL) { // Je-li fronta neprázdná
            // vyjmí proces P z čela fronty
            wakeup(P);         // a probud' P
        }
    }
}
```

# Implementace semaforů

- Záporná hodnota S.value udává, kolik procesů „stojí“ před semaforem
- Fronty před semaforem:
  - Většinou FIFO bez uvažování priorit procesů, jinak vzniká problém se stárnutím
  - Systémy reálného času (RTOS) většinou prioritu uvažují
- Operace wait(S) a post(S) musí být vykonány atomicky
- OS na jednom procesoru nemá problém, OS rozhoduje o přepnutí procesu
- OS na více jádrech:
  - Jádro musí používat atomické instrukce či jiný odpovídající hardwarový mechanismus na synchronizaci skutečného paralelizmu
  - Instrukce xchg, tas, či prefix lock musí umět zamknout sběrnici proti přístupu jiných jader, či zamknout a aktualizovat cache systémem cache snooping

# Mutex

- Mutex – speciální rychlejší semafor, hodnoty pouze 1,0 binární semafor
- Implementace musí zaručit:
- Operace lock() (odpovídá funkci wait() u semaforu) a unlock() (odpovídá funkci post()) musí být atomické stejně jako u semaforů
- Aktivní čekání není plně eliminováno, je ale přesunuto z aplikacní úrovně (kde mohou být kritické sekce dlouhé) do úrovně jádra OS pro implementaci atomicity operací se semafory
- Mutex definuje koncept "vlastníka mutexu" a díky tomu jej lze například zamykat rekurzivně z jednoho vlákna nebo lze implementovat mechanismus pro zabránění uváznutí.

Užití:

```
void *fce(void *n) {  
    int i;  
    for (i=0; i<100000; i++) {  
        pthread_mutex_lock(&mutex);  
        a+=1;  
        pthread_mutex_unlock(&mutex);  
    }  
    pthread_exit(NULL);  
}
```

# Producent – konzument

Tři semafory

- **mutex** s iniciální hodnotou 1 – pro vzájemné vyloučení při přístupu do sdílené paměti
- **used** – počet položek v poli – inicializován na hodnotu 0
- **free** – počet volných položek – inicializován na hodnotu BUF\_SIZE

```
void producer() {
    while (1) { /* Vygeneruj položku do proměnné nextProduced */
        sem_wait(&free);
        sem_wait(&mutex);
        buffer [in] = nextProduced; in = (in + 1) % BUF_SZ;
        sem_post(&mutex);
        sem_post(&used);
    }
}

void consumer() {
    while (1) {
        sem_wait(&used);
        sem_wait(&mutex);
        nextConsumed = buffer[out]; out = (out + 1) % BUF_SZ;
        sem_post(&mutex);
        sem_post(&free);
        /* Zpracuj položku z proměnné nextConsumed */
    }
}
```

# Producent – konzument jen s mutexy

Pro korektní uspání potřebujeme podmínkové proměnné

- čekání na podmínu se provádí funkcí int  
`pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);`
- časově omezené čekání na podmínu se provádí funkcí int  
`pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);`
- při čekání na podmínu se spojí podmínka s mutexem, který se čekáním uvolní
- probuzení vlákna čekajícího na podmínu se provede funkcí int  
`pthread_cond_signal(pthread_cond_t *cond);`
- probuzení všech vláken čekajících na konkrétní podmínu se provede funkcí int `pthread_cond_broadcast(pthread_cond_t *cond);`
- při probuzení vlákna, čekajícího na podmínu se opět mutex obsadí (tedy vlákno počká na jeho uvolnění)
- čekání na podmínu musí být vždy uvnitř kritické sekce mutexu
- probuzení cizího vlákna nastavením podmínky, by mělo být také uvnitř kritické sekce mutexu

# Prodmínkové proměnné - simulace semaforů

```

void sem_wait(struct sem *s) {
    pthread_mutex_lock( &s->mutex );
    while (s->cnt<=0) {
        pthread_cond_wait(&s->cond, &s->mutex);
    }
    s->cnt--;
    pthread_mutex_unlock( &s->mutex );
}

void sem_post(struct sem *s) {
    pthread_mutex_lock( &s->mutex );
    s->cnt++;
    pthread_cond_signal( &s->cond );
    pthread_mutex_unlock( &s->mutex );
}

void sem_init(struct sem *s, int val) {
    s->cnt = val;
    pthread_mutex_init(&s->mutex, NULL);
    pthread_cond_init(&s->cond, NULL);
}

```

```

void *prod(void *n) {
    int i;
    for (i=0; i<30; i++) {
        sem_wait(&empty);
        pthread_mutex_lock(&buf_mutex);
        buf[wr_ptr]=i;
        wr_ptr++; wr_ptr%=10;
        pthread_mutex_unlock(&buf_mutex);
        sem_post(&full);
    }
    return NULL;
}

void *cons(void *n) {
    int i;
    for (i=0; i<30; i++) {
        sem_wait(&full);
        pthread_mutex_lock(&buf_mutex);
        rd_ptr++; rd_ptr%=10;
        pthread_mutex_unlock(&buf_mutex);
        sem_post(&empty);
    }
    return NULL;
}

```

# Prodmínkové proměnné

```

void *producer(void *i) {
    char str[256], *s;
    int wr_ptr=0;

    while ((s=fgets(str, 250, stdin))!=NULL) {
        pthread_mutex_lock(&mutex);
        while (glob_free<=0) {
            printf("Wait full\n");
            pthread_cond_wait(&full, &mutex);
        }
        memcpy(global[wr_ptr], s, 256);
        glob_free--;
        pthread_cond_signal(&empty);
        pthread_mutex_unlock(&mutex);

        wr_ptr=(wr_ptr+1)%8;
    }
    glob_end=1;
    pthread_mutex_lock(&mutex);
    pthread_cond_signal(&empty);
    pthread_mutex_unlock(&mutex);
    return NULL;
}

void *consumer(void *i) {
    int rd_ptr=0;
    while (!glob_end) {
        pthread_mutex_lock(&mutex);
        while(glob_free>=8 && !glob_end) {
            printf("Wait empty\n");
            pthread_cond_wait(&empty, &mutex);
        }
        if (glob_free<8) {
            printf("Zadano: %s\n", global[rd_ptr]);
            glob_free++;
        }
        pthread_cond_signal(&full);
        pthread_mutex_unlock(&mutex);
        rd_ptr=(rd_ptr+1)%8;
        sleep(1);
    }
    return NULL;
}

```

# Čtenáři a písáři

- Úloha: Několik procesů přistupuje ke společným datům
- Některé procesy data jen čtou – čtenáři
- Jiné procesy potřebují data zapisovat – písáři
- Souběžné operace čtení mohou čtenou strukturu sdílet – libovolný počet čtenářů může jeden a tentýž zdroj číst současně
- Operace zápisu musí být exklusivní, vzájemně vyloučená s jakoukoli jinou operací (zápisovou i čtecí)
  - v jednom okamžiku smí daný zdroj modifikovat nejvýše jeden písář
  - Jestliže písář modifikuje zdroj, nesmí ho současně číst žádný čtenář
- Dva možné přístupy
  - Přednost čtenářů
    - Žádný čtenář nebude muset čekat, pokud sdílený zdroj nebude obsazen písářem. Jinak řečeno: Kterýkoliv čtenář čeká pouze na opuštění kritické sekce písářem.
    - Písáři mohou stárnout
  - Přednost písářů
    - Jakmile je některý písář připraven vstoupit do kritické sekce, čeká jen na její uvolnění (čtenářem nebo písářem). Jinak řečeno: Připravený písář předbíhá všechny připravené čtenáře.
    - Čtenáři mohou stárnout

# Priorita čtenářů

## ■ Sdílená data

- semaphore wrt, readcountmutex;
- int readcount

## ■ Inicializace

- wrt = 1; readcountmutex = 1; readcount = 0;

Písář:

```
sem_wait(wrt);
// písář modifikuje zdroj
sem_post(wrt);
```

Čtenář:

```
sem_wait(readcountmutex);
readcount++;
if (readcount==1) sem_wait(wrt);
sem_post(readcountmutex);
```

*// čtení sdíleného zdroje*

```
sem_wait(readcountmutex);
readcount--;
if (readcount==0) sem_post(wrt);
sem_post(readcountmutex);
```

# Priorita písářů

## ■ Sdílená data

- semaphore wrt, rdr, readcountmutex, writecountmutex;
- int readcount, writecount;

## ■ Inicializace

- wrt = 1; rdr = 1; readcountmutex = 1; writecountmutex = 1;
- readcount = 0; writecount = 0;

Písář:

```
sem_wait(writecountmutex);
writecount++;
if (writecount==1) sem_wait(rdr);
sem_post(writecountmutex);
sem_wait(wrt);
    // písář modifikuje zdroj
sem_post(wrt);
sem_wait(writecountmutex);
writecount--;
if (writecount==0) sem_post(rdr);
sem_post(writecountmutex);
```

Čtenář:

```
sem_wait(rdr);
sem_wait(readcountmutex);
readcount++;
if (readcount == 1) sem_wait(wrt);
sem_post(readcountmutex);
sem_post(rdr);
    // čtení sdíleného zdroje
sem_wait(readcountmutex);
readcount--;
if (readcount == 0) sem_post(wrt);
sem_post(readcountmutex);
```

# Monitor

- Monitor je synchronizační nástroj vyšší úrovně
- Umožňuje bezpečné sdílení libovolného datového typu
- Na rozdíl od semaforů, monitor explicitně definuje která data jsou daným monitorem chráněna
- Monitor je jazykový konstrukt v jazycích „pro paralelní zpracování“
- Podporován např. v Concurrent Pascal, Modula-3, C#, ...
- V Javě může každý objekt fungovat jako monitor (viz metoda Object.wait() a klíčové slovo synchronized)
- Procedury definované jako monitorové procedury se vždy vzájemně vyučují

```
monitor monitor_name {  
    int i;                      // Deklarace sdílených proměnných  
    void p1(...) { ... }        // Deklarace monitorových procedur  
    void p2(...) { ... }  
    {  
        // inicializační kód  
    }  
}
```

# Synchronizace v Javě

- Java používá pro synchronizaci Monitor
- Uživatel si může nadefinovat semafor následovně:

```
public class CountingSemaphore {  
    private int signals = 1;  
  
    public synchronized void sem_wait() throws InterruptedException  
        while(this.signals <= 0) wait();  
        this.signals--;  
    }  
  
    public synchronized void sem_post() {  
        this.signals++;  
        this.notify();  
    }  
}
```

Případně lze použít i efektivnější java.util.concurrent.Semaphore

# Spin-lock

- Spin-lock je obecný (čítající) semafor, který používá aktivní čekání místo blokování
- Blokování a přepínání mezi procesy či vlákny by bylo časově mnohem náročnější než ztráta strojového času spojená s krátkodobým aktivním čekáním
- Používá se ve víceprocesorových systémech pro implementaci krátkých kritických sekcí
- Typicky uvnitř jádra
- Např. při obsluze přerušení, kde není možné blokování (přerušení není součástí žádného procesu, jedná se o hardwarový koncept)
- Další použití je pro krátké kritické sekce, např. zajištění atomicity operací se semafory (ale to se většinou řeší efektivnějšími atomickými instrukcemi)
- Užito např. v multiprocesorových Windows 2k/XP/7 i Linuxu

# B4B35OSY: Operační systémy

## Lekce 5. Meziprocesní komunikace

Petr Štěpán  
[stepan@fel.cvut.cz](mailto:stepan@fel.cvut.cz)



10. září, 2020

# Outline

- 1 Uváznutí – Deadlock**
  
- 2 Meziprocesní komunikace**

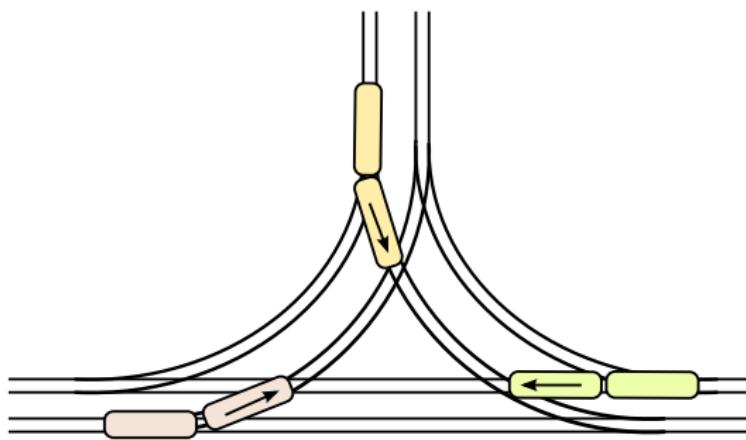
# Obsah

1 Uváznutí – Deadlock

2 Meziprocesní komunikace

# Deadlock v životě

- Karlovo náměstí – tramvaje většinou jedou jen dopředu
- Zdroje jsou křížení tramvajových kolejí
- Aby tramvaj projela, musí použít dva zdroje na své cestě

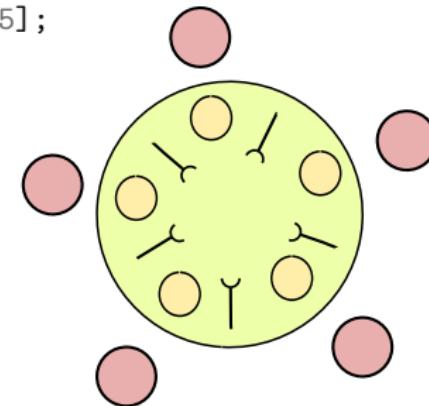


- "It takes money to make money" – anglické přísloví
- K získání kvalitního zaměstnání je potřeba kvalitní praxe, kvalitní praxi získáte pouze v kvalitním zaměstnání

# Večeřící filozofové

Sdílená data

```
/* Inicializace */
semaphore chopStick[ ] = new Semaphore[5];
for(i=0; i<5; i++) chopStick[i] = 1;
/* Implementace filozofa i: */
do {
    chopStick[i].wait;
    chopStick[(i+1) % 5].wait;
    eating();           // Teď jí
    chopStick[i].signal;
    chopStick[(i+1) % 5].signal;
    thinking();         // A teď přemýšlí
} while (TRUE) ;
```



- Možné ochrany proti uváznutí
  - Zrušení symetrie úlohy
  - Jeden filozof bude levák a ostatní praváci (levák zvedá vidličky opačně)
  - Jídlo se n filozofům podává v jídelně s  $n+1$  židlemi
  - Filozof smí uchopit vidličku jen, když jsou obě volné a uchopí obě najednou
  - Příklad obecnějšího řešení – tzv. skupinového zamykání prostředků

# Časově závislé chyby

- Příklad časově závislé chyby
- Procesy  $P_1$  a  $P_2$  spolupracují za použití mutexů A a B

Proces  $P_1$

```
pthread_mutex_lock(&mutex_A);  
pthread_mutex_lock(&mutex_B);  
a+=b;  
pthread_mutex_unlock(&mutex_B);  
pthread_mutex_unlock(&mutex_A);
```

Proces  $P_2$

```
pthread_mutex_lock(&mutex_B);  
pthread_mutex_lock(&mutex_A);  
b+=a;  
pthread_mutex_unlock(&mutex_A);  
pthread_mutex_unlock(&mutex_B);
```

- Deadlock nastane pouze v situaci, že proces  $P_1$  získá mutex A a proces  $P_2$  získá mutex B
- Nebezpečnost takových chyb je v tom, že vznikají jen zřídka kdy za náhodné souhry okolností
- Jsou tudíž fakticky neodladitelné

# Coffmanovi podmínky

Coffman formuloval čtyři podmínky, které musí platit současně, aby uváznutí mohlo vzniknout

## 1 Vzájemné vyloučení, Mutual Exclusion

- sdílený zdroj může v jednom okamžiku používat nejvýše jeden proces

## 2 Postupné uplatňování požadavků, Hold and Wait

- proces vlastnící alespoň jeden zdroj potřebuje další, ale ten je vlastněn jiným procesem, v důsledku čehož bude čekat na jeho uvolnění

## 3 Nepřipouští se odnímání zdrojů, No preemption

- zdroj může uvolnit pouze proces, který ho vlastní, a to dobrovolně, když již zdroj nepotřebuje

## 4 Zacyklení požadavků, Circular wait

- Existuje posloupnost čekajících procesů  $P_0, P_1, \dots, P_k, P_0$  takových, že  $P_0$  čeká na uvolnění zdroje drženého  $P_1$ ,  $P_1$  čeká na uvolnění zdroje drženého  $P_2, \dots, P_{k1}$  čeká na uvolnění zdroje drženého  $P_k$ , a  $P_k$  čeká na uvolnění zdroje drženého  $P_0$ .

- V případě jednoinstančních zdrojů splnění této podmínky značí, že k uváznutí již došlo.

# Co dělat?

Existují čtyři přístupy

- Zcela ignorovat hrozbu uváznutí
  - Pštrosí algoritmus – strč hlavu do píska a předstírej, že se nic neděje
  - Používá většina současných OS včetně většiny implementací UNIXů
  - Linux se snaží o prevenic deadlocku uvnitř jádra, neovlivňuje ale použití deadlocků v uživatelských programech
- Prevence uváznutí
  - Pokusit se přijmout taková opatření, aby se uváznutí stalo vysoce nepravděpodobným
  - Ale pozor! Pokud víme, že k uváznutí může dojít, ale jen s malou pravděpodobností, dojde k němu, když se to hodí nejméně
- Vyhýbání se uváznutí
  - Zajistit, že k uváznutí nikdy nedojde
  - Prostředek se nepřidělí, pokud by hrozilo uváznutí
  - hrozí stárnutí
- Detekce uváznutí a následná obnova
  - Uváznutí se připustí, detekuje se jeho vznik a zajistí se obnova stavu před uváznutím

# Prevence uváznutí

- Konzervativní politikou se omezuje přidělování prostředků
- Přímá metoda – plánovat procesy tak, aby nevznikl cyklus v přidělování prostředků
  - Vzniku cyklu se brání tak, že zdroje jsou očíslovány a procesy je smějí alokovat pouze ve vzrůstajícím pořadí čísel zdrojů
  - Nerealistické – zdroje vznikají a zanikají dynamicky
  - Často ale stačí uvažovat třídy zdrojů (LOCKDEP v jádře Linuxu – podobné jako alg. vyhýbání se uváznutí dále)
- Nepřímé metody (narušení některé Coffmanovy podmínky)
  - Eliminace potřeby vzájemného vyloučení
    - Nepoužívat sdílené zdroje, virtualizace (spooling) periferií
    - Mnoho činností však sdílení nezbytně potřebuje ke své funkci
  - Eliminace postupného uplatňování požadavků
    - Proces, který požaduje nějaký zdroj, nesmí dosud žádný zdroj vlastnit
    - Všechny prostředky, které bude kdy potřebovat, musí získat naráz
    - Nízké využití zdrojů
  - Připustit násilné odnímání přidělených zdrojů (preempce zdrojů)
    - Procesu žádajícímu o další zdroj je dosud vlastněný prostředek odňat
    - To může být velmi rizikantní – zdroj byl již modifikován
    - Proces je reaktivován, až když jsou všechny potřebné prostředky volné
    - Metoda inkrementálního zjišťování požadavků na zdroje – nízká průchodnost
    - Metody prevence uváznutí aplikované za běhu způsobí výrazný pokles průchodnosti systému

# Vyhýbání se uváznutí

- Základní problém:
  - Systém musí mít dostatečné apriorní informace o požadavcích procesů na zdroje
- Nejčastěji se požaduje, aby každý proces udal maxima počtu prostředků každého typu, které bude za svého běhu požadovat
- Algoritmus:
  - Dynamicky se zjišťuje, zda stav subsystému přidělování zdrojů zaručuje, že se procesy v žádném případě nedostanou do cyklu
  - Stav systému přidělování zdrojů je popsán
    - Počtem dostupných a přidělených zdrojů každého typu a
    - Maximem očekávaných žádostí procesů
  - Stav může být bezpečný nebo nebezpečný

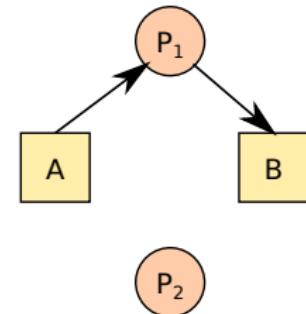
# Vyhýbání se uváznutí

- Systém je v bezpečném stavu, existuje-li „bezpečná posloupnost procesů“
  - Posloupnost procesů  $P_0, P_1, \dots, P_n$  je bezpečná, pokud požadavky každého  $P_i$  lze uspokojit právě volnými zdroji a zdroji vlastněnými všemi  $P_k$ ,  $k < i$ 
    - Pokud nejsou zdroje požadované procesem  $P_i$  volné, pak  $P_i$  bude čekat dokud se všechny  $P_k$  nedokončí a nevrátí přidělené zdroje
    - Když  $P_{i-1}$  skončí, jeho zdroje může získat  $P_i$ , proběhnout a jím vrácené zdroje může získat  $P_{i+1}$ , atd.
- Je-li systém v bezpečném stavu (safe state) k uváznutí nemůže dojít. Ve stavu, který není bezpečný (unsafe state), přechod do uváznutí hrozí
- Vyhýbání se uváznutí znamená:
  - Plánovat procesy tak, aby systém byl stále v bezpečném stavu
  - Nespouštět procesy, které by systém z bezpečného stavu mohly vyvést
  - Nedopustit potenciálně nebezpečné přidělení prostředku

# Model uváznutí

RAG – Resource allocation graph

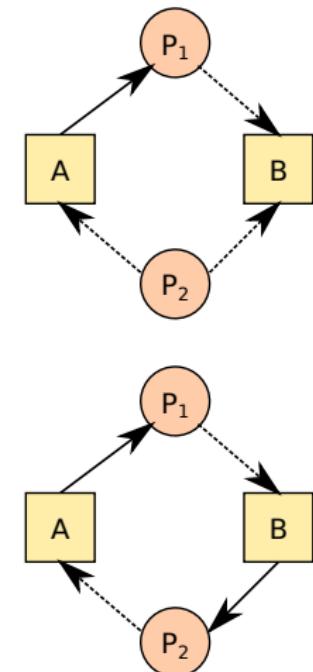
- Typy prostředků (zdrojů)  $R_1, R_2, \dots, R_m$ 
  - např. datové struktury, V/V zařízení, ...
- Každý typ prostředku  $R_i$  má  $W_i$  instancí
  - např. máme 4 síťové karty a 2 CD mechaniky
  - často  $W_i = 1$  tzv. jednoinstanční prostředky – např. mutex
- Každý proces používá potřebné zdroje podle schématu
  - žádost – request, wait
  - používání prostředku po konečnou dobu (kritická sekce)
  - uvolnění (navrácení) – release, signal
- žádost o zdroj značí hrana od procesu  $P_i$  ke zdroji  $R_j$
- přidělený zdroj značí hrana od zdroje  $R_j$  k procesu  $P_i$



# Vyhýbání uváznutí – jednoinstanční zdroje

Potřebujeme znát budoucnost:

- Do RAG se zavede „nároková hrana“
  - Nároková hrana  $P_i \rightarrow R_j$  značí, že někdy v budoucnu bude proces  $P_i$  požadovat zdroj  $R_j$
  - V RAG hrana vede stejným směrem jako požadavek na přidělení, avšak kreslí se čárkovaně
- Nároková hrana se v okamžiku vzniku žádosti o přidělení převede na požadavkovou hranu
  - Když proces zdroj získá, požadavková hrana se změní na hranu přidělení
  - Když proces zdroj vrátí, hrana přidělení se změní na požadavkovou hranu
- Převod požadavkové hrany v hranu přidělení nesmí v RAG vytvořit cyklus (včetně uvažování nárokových hran)
  - LOCKDEP v Linuxu (systém běží cca 10× pomaleji)



# Bankéřský algoritmus

- Chování odpovědného bankéře:
  - Klienti žádají o půjčky do určitého limitu
  - Bankéř ví, že ne všichni klienti budou svůj limit čerpat současně a že bude půjčovat klientům prostředky postupně
  - Všichni klienti v jistém okamžiku svého limitu dosáhnou, avšak nikoliv současně
  - Po dosažení přislíbeného limitu klient svůj dluh v konečném čase vrátí
  - Příklad:
    - Ačkoliv bankéř ví, že všichni klienti budou dohromady potřebovat 22 jednotek a na celou transakci má jen 10 jednotek, je možné uspokojit postupně všechny klienty

# Bankéřský algoritmus

- Zákazníci přicházející do banky pro úvěr předem deklarují maximální výši, kterou si budou kdy chtít půjčit
- Úvěry v konečném čase splácí
- Bankér úvěr neposkytne, pokud si není jist, že bude moci uspokojit všechny zákazníky (vždy alespoň jednomu zákazníku bude moci půjčit všechny peníze a zákazník je pak vrátí)
- Analogie
  - Zákazník = proces
  - Úvěr = přidělovaný prostředek
- Vlastnosti
  - Procesy musí deklarovat své potřeby předem
  - Proces požadující přidělení může být zablokován
  - Proces vrátí všechny přidělené zdroje v konečném čase
- Nikdy nedojde k uváznutí
  - Proces bude spuštěn jen, pokud bude možno uspokojit všechny jeho požadavky
  - Sub-optimální pesimistická strategie
  - Předpokládá se nejhorší případ

# Bankéřský algoritmus

## Datové struktury

- $n$  ... počet procesů
- $m$  ... počet typů zdrojů
- Vektor  $available[m]$ 
  - $available[j] = k$  značí, že je  $k$  instancí zdroje typu  $R_j$  je volných
- Matice  $max[n, m]$ 
  - povinná deklarace procesů maximálních požadavků
  - $max[i, j] = k$  znamená, že proces  $P_i$  bude během své činnosti požadovat až  $k$  instancí zdroje typu  $R_j$
- Matice  $allocated[n, m]$ 
  - $allocated[i, j] = k$  značí, že v daném okamžiku má proces  $P_i$  přiděleno  $k$  instancí zdroje typu  $R_j$
- Matice  $needed[n, m]$  ( $needed[i, j] = max[i, j] - allocated[i, j]$ )
  - $needed[i, j] = k$  říká, že v daném okamžiku procesu  $P_i$  chybí ještě  $k$  instancí zdroje typu  $R_j$

# Bankéřský algoritmus

## Test bezpečnosti stavu

### 1 Inicializace

- $work[m]$  a  $finish[n]$  jsou pracovní vektory
- Inicializujeme  $work = available$ ;  $finish[i] = false; i = 1, \dots, n$

- 2 Najdi  $i$ , pro které platí ( $finish[i] = false$ )  $\wedge$  ( $needed[i] \leq work[i]$ )
- 3 Pokud takové  $i$  neexistuje, jdi na krok 6
- 4 Simuluj dokončení procesu  $i$ 
  - $work[i] = work[i] + allocated[i]$ ;  $finish[i] = true$ ;
- 5 Pokračuj krokem 2
- 6 Pokud platí  $finish[i] = true$  pro všechna  $i$ , pak stav systému je bezpečný

# Bankéřský algoritmus

- Proces  $P_i$  formuje vektor request:
- $request[j] = k$  znamená, že proces  $P_i$  žádá o  $k$  instancí zdroje typu  $R_j$
- $\text{if}(request[j] \geq needed[i, j])$  proces nárokuje víc než bylo maximum na začátku;
- $\text{if}(request[j] \geq available[j])$  zatím nelze splnit, je nutné počkat na uvolnění zdrojů, proces se uspí;
- Jinak otestuj přípustnost požadavku simulováním přidělení prostředku a pak ověříme bezpečnost stavu:
  - $available[j] = available[j] - request[j]$ ;
  - $allocated[i, j] = allocated[i, j] + request[j]$ ;
  - $needed[i, j] = needed[i, j] - request[j]$ ;
  - Spusť test bezpečnosti stavu
  - Je-li bezpečný, přiděl požadované zdroje
  - Není-li stav bezpečný, pak vrat úpravy „Akce 3“ a zablokuj proces  $P_i$ , neboť přidělení prostředků by způsobilo nebezpečí uváznutím

# Bankéřský algoritmus – příklad

Test bezpečnosti stavu:

Prostředky na začátku:

A	B	C
10	6	6

Maximum:

	A	B	C
$P_1$	8	4	4
$P_2$	2	1	4
$P_3$	6	3	3
$P_4$	5	4	3

Alokace:

	A	B	C
$P_1$	3	1	1
$P_2$	1	0	1
$P_3$	2	1	0
$P_4$	1	3	1

Požadavek = Maximum – Alokace:

	A	B	C
$P_1$	5	3	3
$P_2$	1	1	3
$P_3$	4	2	3
$P_4$	4	1	2

Dostupné prostředky

A	B	C
3	1	3

Hledáme proces, který by mohl být dokončen, požadavek  $\leq$  dostupné prostředky, pouze  $P_2$ .

Po dokončení tohoto procesu, proces uvolní svoje prostředky:

A	B	C
4	1	4

Opět hledáme proces, který by mohl být dokončen, pouze  $P_4$ .

Po dokončení tohoto procesu, proces uvolní svoje prostředky:

A	B	C
5	4	5

Opět hledáme proces, který by mohl být dokončen, nyní lze dokončit  $P_1$  a  $P_3$ . Po dokončení procesu např.  $P_1$ , proces uvolní svoje prostředky:

A	B	C
8	5	6

Nyní lze dokončit  $P_3$ , po dokončení jsou dostupné prostředky stejné jako na začátku:

A	B	C
10	6	6

Všechny procesy skončily, stav je bezpečný.

# Bankéřský algoritmus – příklad

Proces  $P_3$  žádá o 2 zdroje A:

Prostředky na začátku:

A	B	C
10	6	6

Maximum:

	A	B	C
$P_1$	8	4	4
$P_2$	2	1	4
$P_3$	6	3	3
$P_4$	5	4	3

Alokace po simulovaném přidělení prostředků:

	A	B	C
$P_1$	3	1	1
$P_2$	1	0	1
$P_3$	4	1	0
$P_4$	1	3	1

Požadavek = Maximum – Alokace

	A	B	C
$P_1$	5	3	3
$P_2$	1	1	3
$P_3$	2	2	3
$P_4$	4	1	2

Dostupné prostředky:

A	B	C
1	1	3

Hledáme proces, který by mohl být dokončen, požadavek  $\leq$  dostupné prostředky, pouze  $P_2$ .

Po dokončení tohoto procesu, proces uvolní svoje prostředky:

A	B	C
2	1	4

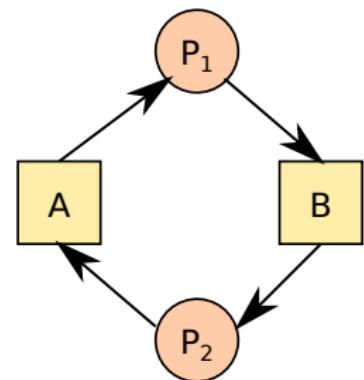
- Nyní nelze dokončit žádný proces, neboť dostupné prostředky nestačí pro dokončení žádného procesu (procesu  $P_1$  chybí 3 zdroje A, 2 zdroje B; procesu  $P_3$  chybí 1 zdroj B, procesu  $P_4$  chybí 2 zdroje A).
- Stav není bezpečný – pokud by všechny procesy chtěli své maximum a teprve potom uvolnili zdroje, pak nastane deadlock.
  - O chování procesů nic nevíme, ale abychom se vyhnuli deadlocku musíme předpokládat nejhorší případ

## Detekce uváznutí

- Strategie připouští vznik uváznutí:
- Uváznutí je třeba detektovat
- Vznikne-li uváznutí, aplikuje se plán obnovy systému
- Aplikuje se zejména v databázových systémech, kde je obnova dotazu běžná

# Detekce uváznutí s RAG

- Případ jednoinstančního zdroje daného typu
  - Udržuje se čekací graf – uzly jsou procesy
  - Periodicky se provádí algoritmus hledající cykly
  - Algoritmus pro detekci cyklu v grafu má složitost  $O(n^2)$ , kde n je počet hran v grafu



# Detekce uváznutí

- Případ více instancí zdrojů daného typu
  - $n$  ... počet procesů
  - $m$  ... počet typů zdrojů
  - Vektor  $available[m]$ 
    - $available[j] = k$  značí, že je  $k$  instancí zdroje typu  $R_j$  je volných
  - Matice  $allocated[n, m]$ 
    - $allocated[i, j] = k$  značí, že v daném okamžiku má proces  $P_i$  přiděleno  $k$  instancí zdroje typu  $R_j$
  - Matice  $request[n, m]$ 
    - Indikuje okamžité požadavky každého procesu:
    - $request[i, j] = k$  znamená, že proces  $P_i$  požaduje dalších  $k$  instancí zdroje typu  $R_j$

# Detekce uváznutí pro více-instanční problémy

Obdoba bankéřského algoritmu (nevíme budoucnost – neznáme maximum):

- $work[m]$  a  $finish[n]$  jsou pracovní vektory
  - 1 Inicializujeme  $work = available$ ;  $finish[i] = false; i = 1, \dots, n$
  - 2 Najdi  $i$ , pro které platí ( $finish[i] = false$ ) and ( $request[i] \leq work[i]$ )
  - 3 Pokud takové  $i$  neexistuje, jdi na krok 6
  - 4 Simuluj dokončení procesu  $i$ 
    - $work[i] += allocated[i];$
    - $finish[i] = true;$
  - 5 Pokračuj krokem 2
  - 6 Pokud platí
    - $finish[i] = false$  pro některé  $i$ , pak v systému došlo k uváznutí.
    - Součástí cyklů ve WG jsou procesy  $P_i$ , kde  $finish[i] == false$
- Algoritmus má složitost  $O(m \cdot n^2)$ ,  $m$  a  $n$  mohou být veliká a algoritmus časově značně náročný

# Detekce uváznutí

Kdy a jak často algoritmus vyvolávat? (Detekce je drahá)

- Jak často bude uváznutí vznikat?
- Kterých procesů se uváznutí týká a kolik jich „likvidovat“?
  - Minimálně jeden v každém disjunktním cyklu v RAG
  - Násilné ukončení všech uvázlých procesů – velmi tvrdé a nákladné
  - Násilně se ukončují dotčené procesy dokud cyklus nezmizí
    - Jak volit pořadí ukončování
    - Jak dlouho už proces běžel a kolik mu zbývá do dokončení
    - Je to proces interaktivní nebo dávkový (dávku lze snáze restartovat)
    - Cena zdrojů, které proces použil
    - Výběr oběti podle minimalizace ceny
    - Nebezpečí stárnutí
    - některý proces bude stále vybíráno jako oběť

# Uváznutí – shrnutí

- Metody popsané jako „prevence uváznutí“ jsou velmi restriktivní
  - ne vzájemnému vyloučení, ne postupnému uplatňování požadavků, preempce prostředků
- Metody „vyhýbání se uváznutí“ nemají dost apriorních informací
  - zdroje dynamicky vznikají a zanikají (např. úseky souborů)
- Detekce uváznutí a následná obnova
  - jsou vesměs velmi drahé – vyžadují restartování aplikací
- Smutný závěr
  - Problém uváznutí je v obecném případě efektivně neřešitelný
  - Existuje však řada algoritmů pro speciální situace – zejména používané v databázových systémech
    - Transakce vědí, jaké tabulky budou používat
    - Praktickým řešením jsou distribuované systémy
    - Minimalizuje se počet sdílených prostředků
    - Nutnost zabývat se uváznutím v uživatelských paralelních a distribuovaných aplikacích

# Obsah

1 Uváznutí – Deadlock

2 Meziprocesní komunikace

# Meziprocesní komunikace

## Přehled meziprocesní komunikace

Název	Anglicky	Standard
Signál	Signal	POSIX
Roura	Pipe	POSIX
Pojmenovaná roura	Named pipe	POSIX
Soubor mapovaný do paměti	Memory-mapped file	POSIX
Sdílená paměť	Shared memory	System V IPC
Semafor	Semaphore	System V IPC
Zasílání zpráv	Message passing	System V IPC
Soket	Socket	Networking

# Signály

- seznámili jste se již na cvičeních
- zaslání jednoduché zprávy (nastavení 1 bitu), která je definována číslem signálu
- příjemcem signálu je pouze proces, odesílatel je buď proces, nebo jádro OS
- obsluha signálů:
  - struct sigaction – sa\_handler, či sa\_sigaction
  - funkce sigaction – připojení funkce k obsluze signálu
- signál se zpracovává asynchroně (nezávisle) na přijímajícím procesu
  - dojde k přepnutí kontextu a spustí se připojená funkce
  - POZOR na kritické sekce se sdílenými proměnnými

# Signály

Použití signálů:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <string.h>

int volatile zalohuj = 0;

void handler(int num) {
    zalohuj = 1;
}

int main() {
    pid_t child = fork();
    if (child == 0) {
        int prace = 30;
        struct sigaction action;
        memset(&action, 0, sizeof(action));
        action.sa_handler = handler;
        if (sigaction(SIGUSR1, &action, NULL) != 0)
            { return 2; }
        while (prace-->0) {
            printf("PRACUJI\n");
            sleep(1);
            if (zalohuj) {
                printf("Ukladam mezivysledek\n");
                zalohuj = 0;
            }
        } else {
            int status;
            while (!waitpid(child, &status, WNOHANG)) {
                sleep(5);
                kill(child, SIGUSR1);
            }
        }
    }
    return 0;
}
```

# Roury

- seznámili jste se již na cvičeních
- zaslání dat mezi procesy systémem FIFO
- vlastně simulovaný neexistující soubor
- může být více příjemců i více zapisujících procesů – vznikají ale problémy stejné jako při psaní a čtení více procesů ze souborů
- použití roury:
  - `int pipe(int [2])` – rodič vytvoří rouru
  - všichni potomci i rodič může do roury zapisovat i číst
  - standardně rouru zavřou všichni, kdo ji nepoužívají a slouží k přesunu dat mezi dvěma procesy

# Roury

Použití roury:

```
int main() {
    int pipef[2];
    if (pipe(pipef)!=0) {return 2;}

    pid_t child = fork();
    if (child == 0) {
        dup2(pipef[1],1);
        close(pipef[0]);
        close(pipef[1]);
        for (int i=0; i<10000; i++) {
            printf("Data %d\n", i);
            fflush(stdout);
        }
    } else { // rodič
        char line[256];
        dup2(pipef[0], 0);
        close(pipef[0]);
        close(pipef[1]);
        while (fgets(line, sizeof(line), stdin))
        {
            printf("Prijata data: %s\n", line);
        }
        wait(&status1);
    }
    return 0;
}
```

# Pojmenované roury

- stejný princip – zaslání data mezi procesy systémem FIFO
- může být více příjemců i více zapisujících procesů – vznikají ale problémy stejné jako při psaní a čtení více procesů ze souborů
- oproti normální rouře ji mohou používat libovolné procesy
- použití pojmenované roury:
  - `mkfifo` – vytvoření roury z příkazové řádky
  - `int mknod(const char *, mode_t, dev_t)` – vytvoří pojmenovanou rouru programem pokud mode využije S\_IFIFO
  - všechny procesy pak mohou rouru využít jako "standardní" soubor

# Pojmenované roury

Použití roury: Producent

```
int main() {
    char line[1000];
    mknod("/tmp/myfifo", S_IFIFO | 0660, 0);
    int fd = open("/tmp/myfifo", O_WRONLY, 0);
    for (int i=0; i<100000; i++) {
        sprintf(line, "Data %i\n", i);
        write(fd, line, strlen(line));
    }
    return 0;
}
```

Konzument

```
int main() {
    char line[1024];
    int fd = open("/tmp/myfifo", O_RDONLY, 0), rd;
    while ((rd=read(fd, line, 1000))>0) {
        line[rd]=0;
        printf("Prijata data: %i %s\n", rd, line);
    }
    return 0;
}
```



# Sdílený soubor mapovaný do paměti

- data lze přenášet mezi procesy pomocí souborů
  - jeden proces zapisuje do souboru, druhý čte ze souboru
- použití normálních souborů je pomalejší, i když je zápis bufferován v paměti
- rychlejší varianta je soubor mapovaný do paměti
- vybraný úsek souboru je mapován přímo do paměti procesu:
  - `mmap(void *addr, size_t delka, int proto, int typ, int fd, off_t posunuti)`
  - vrací adresu, kterou lze použít pro zápis/čtení přímo do/z paměti
  - soubor musí mít alespoň délku, kterou mapujeme do paměti
  - synchronizace je složitější, nejlépe za použití jiného mechanismu
  - Mapování souboru `/dev/zero` = alokace paměti

# Soubor mapovaný do paměti

```
int main() {
    char *shared_mem, buf[256];
    int fd = open("/tmp/mapedfile",
                  O_RDWR | O_CREAT, 0660);
    printf("Open file %i\n", fd);
    for (int i=0; i<1000; i++) {
        sprintf(buf, "Data %03i\n", i);
        buf[9]=0;
        write(fd, buf, 10);
    }
    shared_mem = mmap(NULL, 10000,
                      PROT_READ | PROT_WRITE,
                      MAP_SHARED, fd, 0);
    printf("mmap %p\n", shared_mem);
    close(fd);
    return 0;
}
```

```
int main() {
    char *shared_mem;
    int fd = open("/tmp/mapedfile",
                  O_RDWR | O_CREAT, 0660);
    shared_mem = mmap(NULL, 10000,
                      PROT_READ | PROT_WRITE,
                      MAP_SHARED, fd, 0);
    for (int i=0; i<1000; i++) {
        printf("Ulozeno: %s\n", shared_mem);
        shared_mem+=10;
    }
    close(fd);
    return 0;
}
```

# Sdílená paměť

- velmi podobné, jako soubory mapované do paměti
- soubor je pouze virtuální a nezapisuje se na disk
- po vypnutí počítače se data ve sdílené paměti ztratí
- použití je velmi podobné:
  - `shm_open` – vytvoří virtuální soubor, nebo připojí k existujícímu podle jména
  - `mmap(void *addr, size_t délka, int proto, int typ, int fd, off_t posunutí)`
    - vrací adresu, kterou lze použít pro zápis/čtení přímo z paměti
  - `ftruncate` nebo `write`, kvůli vytvoření místa ve virtuálním souboru
  - synchronizace je složitější, nejlépe za použití jiného mechanismu

# Sdílená paměť – příklad

```
int main() {
    char *shared_mem;
    int fd = shm_open("pamet",
        O_RDWR | O_CREAT | O_TRUNC, 0660);
    ftruncate(fd, 10000);
    shared_mem = mmap(NULL, 10000, PROT_READ
        | PROT_WRITE, MAP_SHARED, fd, 0);
    for (int i=0; i<1000; i++) {
        sprintf(shared_mem, "Data %03i\n", i);
        shared_mem[9]=0;
        shared_mem += 10;
    }
    close(fd);
    shm_unlink("pamet");
    return 0;
}
```

```
int main() {
    char *shared_mem;
    int fd = shm_open("pamet",
        O_RDWR | O_CREAT, 0660);
    shared_mem = mmap(NULL, 10000, PROT_READ
        | PROT_WRITE, MAP_SHARED, fd, 0);
    close(fd);
    for (int i=0; i<1000; i++) {
        printf("Ulozeno: %s\n", shared_mem);
        shared_mem+=10;
    }
    return 0;
}
```

# Pojmenovaný semafor

- podobně jako pojmenovaná roura je možné k němu přistoupit z nového procesu
- semafor se připojuje k již existujícímu souboru
  - pouze identifikace, nic do souboru neukládá
- podobně jako semafory pro vlákna, umožňuje implementovat kritickou sekci, nebo počítat
- použití semaforu:
  - ftok – vytvoří identifikátor (klíč) podle jména souboru
  - semget – připojí/vytvoří semafor ke klíči
  - semctl – nastaví hodnotu semaforu
  - semop – provede operaci (odečtení, nebo přičtení)

# Semaphore System V a sdílená paměť

```

int main() {
    key_t s_key;
    union semun {
        int val;
        struct semid_ds *buf;
        ushort array [1];
    } sem_attr;
    struct sembuf asem;
    int buffer_count_sem, spool_signal_sem;
    char *shared_mem;

/* key identifikuje semafor */
if ((s_key = ftok ("/tmp/free", 'a')) == -1)
{ perror ("ftok"); exit (1); }
if ((buffer_count_sem = semget (s_key, 1,
    0660 | IPC_CREAT)) == -1)
{ perror ("semget"); exit (1); }
sem_attr.val = 10; // nastav na velikost bufferu
if (semctl (buffer_count_sem, 0, SETVAL, sem_attr)
    == -1) { perror (" semctl SETVAL "); exit (1); }
/* key druhého semaforu */
if ((s_key = ftok ("/tmp/data", 'a')) == -1) {
    perror ("ftok"); exit (1);
}
if ((spool_signal_sem = semget (s_key, 1,
    0660 | IPC_CREAT)) == -1) {
    perror ("semget"); exit (1);
}
}

```

```

sem_attr.val = 0; // inicializace na 0
if (semctl (spool_signal_sem, 0,
    SETVAL, sem_attr) == -1)
{ perror (" semctl SETVAL "); exit (1); }
int fd = shm_open("pamet", O_RDWR
    | O_CREAT | O_TRUNC, 0660);
ftruncate(fd, 1000);
shared_mem = mmap(NULL, 1000, PROT_READ
    | PROT_WRITE, MAP_SHARED, fd, 0);
close(fd);
asem.sem_num = 0;
asem.sem_flg = 0;
for (int i=0; i<50; i++) {
    asem.sem_op = -1;
    if (semop (buffer_count_sem, &asem, 1) == -1)
    { perror ("semop: buffer_count_sem"); exit (1); }
    sprintf(shared_mem, "Data %03i\n", i);
    printf( "Data %03i %p\n", i, shared_mem);
    shared_mem[9]=0;
    shared_mem += 10;
    if (i%10==9) {
        shared_mem-=100;
    }
    asem.sem_op = 1;
    if (semop (spool_signal_sem, &asem, 1) == -1)
    { perror ("semop: spool_signal_sem"); exit (1); }
}
return 0;
}

```

# Semaphore System V a sdílená paměť

```

int main() {
    key_t s_key;
    union semun {
        int val;
        struct semid_ds *buf;
        ushort array [1];
    } sem_attr;
    struct sembuf asem;
    int buffer_count_sem, spool_signal_sem;

/* pouzij stejny semafor jako producent */
if ((s_key = ftok ("/tmp/free", 'a')) == -1) {
    perror ("ftok");
    exit (1);
}
if ((buffer_count_sem = semget (s_key, 1,
    0660 | IPC_CREAT)) == -1) {
    perror ("semget");
    exit (1);
}

/* pouzij stejny semafor jako producent */
if ((s_key = ftok ("/tmp/data", 'a')) == -1) {
    perror ("ftok");
    exit (1);
}
if ((spool_signal_sem = semget (s_key, 1,
    0660 | IPC_CREAT)) == -1) {
    perror ("semget");
    exit (1);
}

char *shared_mem;
int fd = shm_open("pamet", O_RDWR |
    O_CREAT, 0660);
ftruncate(fd, 1000);
shared_mem = mmap(NULL, 1000, PROT_READ
    | PROT_WRITE, MAP_SHARED, fd, 0);
close(fd);
asm.sem_num = 0;
asm.sem_flg = 0;
for (int i=0; i<50; i++) {
    asem.sem_op = -1;
    if (semop (spool_signal_sem, &asm, 1) == -1) {
        perror ("semop: spool_signal_sem");
        exit (1);
    }
    printf("Ulozeno: %s, %p\n", shared_mem, shared_mem);
    shared_mem += 10;
    if (i%10==9) {
        shared_mem=100;
        sleep(1);
    }
    asem.sem_op = 1;
    if (semop (buffer_count_sem, &asm, 1) == -1) {
        perror ("semop: buffer_count_sem");
        exit (1);
    }
}
close(fd);
return 0;
}

```

# Fronta zpráv

- zprávy jsou zasílány a vyzvedávány do/z fronty zpráv identifikovaných libovolným souborem
- podobně jako pojmenovaná roura a semafor je možné k němu přistoupit z nového procesu
- zprávy mají povinně typ, podle kterého je možné vybírat z fronty zpráv pouze zprávy zadанého typu
- použití fronty zpráv:
  - msgget – vytvoří virtuální frontu zpráv, nebo připojí k existující frontě podle jména souboru zadlého jeho klíčem
    - nutné vytvořit si vlastní strukturu zpráv, která jako první obsahuje long – typ zprávy
  - msgsnd – zaslání zprávy, pozor délka zprávy je délka struktury zmenšená o velikost long – typ zprávy
  - msgrcv – přijmutí zprávy zadlého typu
  - msgctl – odstranění fronty zpráv

# Fronty zpráv

```

struct my_msg {
    long mtype;
    int len;
    char txt[10];
};

int main() {
    key_t s_key;
    int msg_id;
    struct my_msg msg;
    if ((s_key = ftok ("/tmp", 'a')) == -1)
        { perror ("ftok"); exit (1); }
    if ((msg_id = msgget(s_key, 0660 | IPC_CREAT))
        == -1) { perror ("msgget"); exit (1); }
    for (int i=0; i<50; i++) {
        msg.mtype=1;
        msg.len = 10;
        sprintf(msg.txt, "Data %03i\n", i);
        msg.txt[9]=0;
        if (msgsnd(msg_id, &msg, sizeof(msg)-sizeof(long),0)
            == -1) { perror ("msgsnd"); exit (1); }
    }
    if (msgrcv(msg_id, &msg, sizeof(msg)-sizeof(long), 2, 0)
        == -1) { perror ("msgrcv"); exit (1); }
    printf("Prijato: %s\n", msg.txt);
    if (msgctl(msg_id, IPC_RMID, 0) == -1)
        { perror ("msgctl"); exit (1); }
    return 0;
}
int main() {
    key_t s_key;
    int msg_id;
    struct my_msg msg;

    if ((s_key = ftok ("/tmp", 'a')) == -1) {
        perror ("ftok"); exit (1);
    }
    if ((msg_id = msgget(s_key, 0660 | IPC_CREAT))
        == -1) { perror ("msgget"); exit (1); }
    for (int i=0; i<50; i++) {
        if (msgrcv(msg_id, &msg, sizeof(msg)-sizeof(long), 1, 0)
            == -1) { perror ("msgrcv"); exit (1); }
        printf("Prijato: %s\n", msg.txt);
    }
    msg.mtype=2;
    msg.len = 10;
    sprintf(msg.txt, "Koncime\n");
    msg.txt[9]=0;
    if (msgsnd(msg_id, &msg, sizeof(msg)-sizeof(long), 0)
        == -1) { perror ("msgsnd"); exit (1); }
    return 0;
}

```

# B4B350SY: Operační systémy

## Lekce 6. Správa paměti

Petr Štěpán  
[stepan@fel.cvut.cz](mailto:stepan@fel.cvut.cz)



7. listopadu, 2018

# Outline

1 Správa paměti

2 Virtualizace paměti

# Obsah

1 Správa paměti

2 Virtualizace paměti

# Názvosloví

## ■ FAP

- fyzický adresní prostor
- skutečná paměť počítače – RAM
- velikost závisí na možnostech základní desky a na osazených paměťových modulech

## ■ LAP

- logický adresní prostor
- někdy také virtuální paměť
- velikost záleží na architektuře CPU
  - 16 bitová adresace – 64 KiB
  - 20 bitová adresace – 1 MiB
  - 32 bitová adresace – 4 GiB
  - 64(48) bitová adresace – 16 PiB

# Počítače bez správy paměti

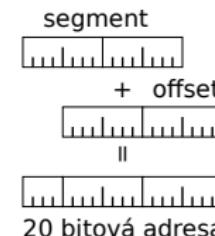
- Výhody systému bez správy paměti:
  - rychlosť přístupu do paměti
  - jednoduchost implementace
  - lze používat i bez operačního systému – robustnosť
- Nevýhody systému bez správy paměti
  - Nelze kontrolovať přístup do paměti  
(kdokoli může cokoli v paměti přepsat)
  - Omezení paměti vlastnostmi HW
- Použití
  - Historické počítače
    - Osmibitové počítače (procesory Intel 8080, Z80, apod.)
    - 8bitová datová sběrnice, 16bitová adresová sběrnice, možnost využít maximálně 64 kB paměti
  - Programovatelné mikrokontrolery
  - Řídicí počítače – embedded
  - V současné době již jen ty nejjednodušší řídicí počítače 8/16-bitové(např. Atmel Xomega)

# Jednoduché segmenty

## Jednoduché segmenty – Intel 8086

- Procesor 8086 má 16bitovou datovou sběrnici a registry
- Procesor má 20 bitů adresové sběrnice.
- 20 bitů je ale problém. Co s tím?
- Řešením jsou jednoduché segmenty:

- Procesor 8086 má 4 tzv. segmentové registry
- Adresa je tvořena adresou segmentu 16 bitů a
- adresou uvnitř segmentu (offset) 16 bitů.
- Výsledná FA se tvoří podle pevného pravidla:
- $adr = (\text{segment} \ll 4) + \text{offset}$

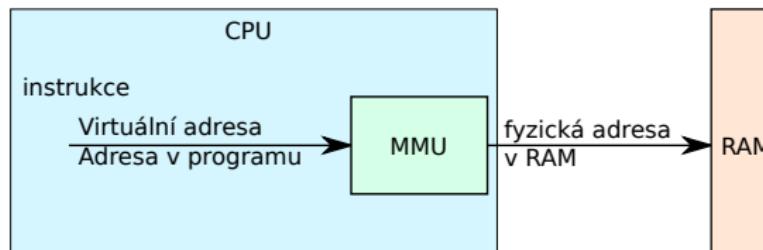


- Prostředek, jak používat větší paměť než dovoluje systém
- Využívá se i v současnosti u jednoduchých 16-bitových procesorů (např. Infineon xc167)
  - někdy se hovoří o mapování
- zavádí dva druhy adres:
  - near – uvnitř segmentu, pouze 16bitový ukazatel
  - far – mezi segmenty, 16bitový ukazatel a 16 bitů číslo segmentu

# Segmentace

## Skutečné segmenty – Intel 80286

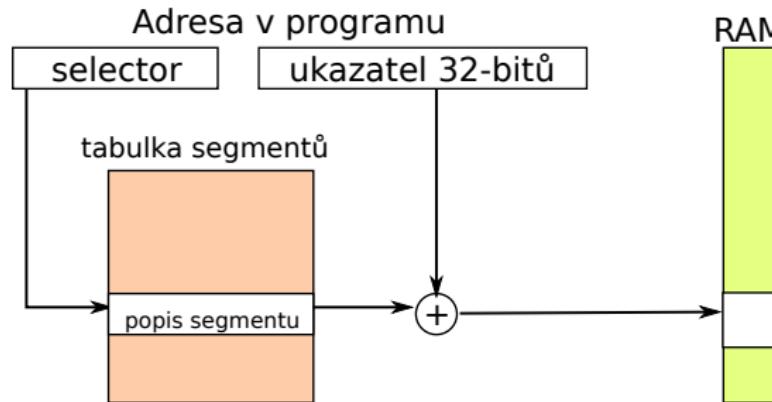
- První procesor s MMU na čipu
  - MMU – memory management unit – převádí adresu (16 bit selektor, 16 bit offset) na 24 bitů adresu ve fyzické paměti



- Program je kolekce segmentů
- Každý segment má svůj logický význam:
  - hlavní program, procedura, funkce,
  - objekt a jeho metoda, proměnné, pole, ...

# Segmenty – Intel 80286

- Základní úkol – převést adresu typu (segment selector, offset) na adresu FAP
  - CS, DS, SS, ES – code, data, stack, extra segment selector 16 bitů
    - bit 0–1 RPL – request privilege level – úroveň ochrany
    - bit 2 TI – 0 – global descriptor (patří všem procesům), 1 – local descriptor (jen pro jeden proces)
    - bit 3–15 – index v tabulce



# Segmenty – Intel 80286

- Tabulka segmentů – Segment table (ST) – Zobrazení 2-D (segment, offset) LAP do 1-D (adresa) FAP
- ST je uložena v normální paměti RAM, informace vybraných segmentech CS, DS, ES, SS je uložena uvnitř CPU
  - Registr L/GDT – Local/global descriptor table 24-bitů – umístění tabulky segmentů v paměti
  - Registr LL/GDT – limit Local/global descriptor table 16 bitů – počet segmentů procesu, velikost tabulky
- Položka ST (64 bitů):
  - base – 24 bitů – počáteční adresa umístění segmentu ve FAP,
  - limit – 16 bitů – délka segmentu (max. 64 KiB)
  - práva – 8 bitů – P – present, DPL – descriptor privilege level, S – Segment descriptor (system/user) oprávnění ke zápisu, E – executable, ED Expansion direction (>limit, <limit), w – Writeable, A – accessed
  - rezerva – 8 bitů – pro 386 rozšíření
    - 386 modifikuje segmenty na 20 bitů limit, 32 bitů adresa
    - G – velikost segmentu je v bajtech, nebo v  $2^{12}$  tj. 4KiB – maximální velikost  $2^{32}$
- vzdálené skoky – lcall – long call, int – i změna segmentu, lret, iret – vzdálený návrat

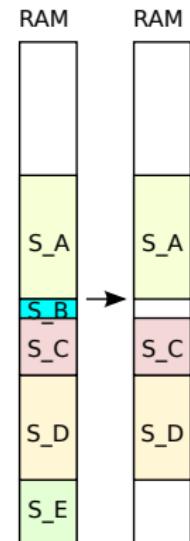
# Segmentace – vlastnosti

- Výhody segmentace
  - Segment má délku uzpůsobenou skutečné potřebě
    - minimum vnitřní fragmentace
    - Lze detekovat přístup mimo segment, který způsobí chybu segmentace – výjimku typu „segmentation fault“
  - Lze nastavovat práva k přístupu do segmentu
    - Operační systém požívá větší ochrany než aplikační proces
    - Uživatel nemůže ohrozit operační systém
  - Lze pohybovat s daty i programem v fyzické paměti
    - posun počátku segmentu je pro aplikační proces neviditelný a nedetekovatelný
- Nevýhody segmentace
  - Alokace segmentů v paměti je netriviální úloha
    - Segmenty mají různé délky. Při běhu více procesů se segmenty ruší a vznikají nové.
    - Problém s externí fragmentací
  - Režie při přístupu do paměti
    - Převod na lineární adresu se opírá o tabulku segmentů a ta je také v paměti
    - Při změně segmentového registru – nutné načíst položku z tabulky
    - Častá změna segmentů (po pář instrukcí) – časově náročná

# Alokace segmentů

Obecný problém – alokace bloků paměti – umístění segmentu v RAM

- "Díra" = blok neobsazené paměti
- Segmentu se přiděluje díra, která jeho požadavek uspokojí
  - tím může vzniknout další malá díra
- Díry jsou roztroušeny po FAP
- Evidenci o dírách a obsazených místech udržuje jádro OS
- Kde přidělit oblast délky  $n$ , když je volná paměť rozmístěna ve více souvislých nesousedních sekčích?
  - First fit – první volná oblast dostatečné velikosti – rychlé, nejčastější
  - Best fit – nejmenší volná oblast dostatečné velikosti – neplýtvá velkými dírami, mohou vznikat mini-díry
  - Worst fit – největší volná oblast – zanechává velké volné díry vhodné pro další alokaci



# Fragmentace

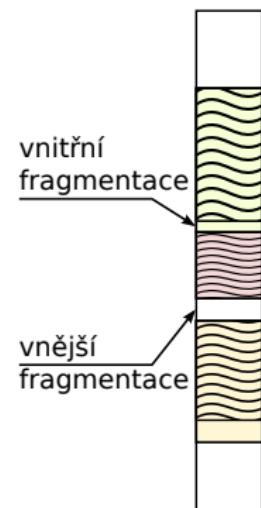
Obecný problém nevyužitelného prostoru

## ■ Externí (vnější) fragmentace

- Celkové množství volné paměti je sice dostatečné, aby uspokojilo požadavek procesu, avšak prostor není souvislý, takže ho nelze přidělit
- Existence mnoha malých dér

## ■ Interní (vnitřní) fragmentace

- Přidělená díra v paměti je o málo větší než potřebná, avšak zbytek je tak malý, že ho nelze využít
- Redukce externí fragmentace pomocí setřásání
- Přesouvají se obsahy úseků paměti s cílem vytvořit (jeden) velký souvislý volný blok
- Použitelné pouze při dynamické relokaci
- Při absolutních adresách v paměti by bylo nutno přepočítat a upravit všechny adresy v instrukcích
- Problém s I/O: S vyrovnávacími pamětími plněnými z periferií (zejména přes DMA) nelze kdykoli hýbat, umisťují se proto do prostoru JOS



# Stránkování

## Stránkování – Intel 80386

- Procesor 386 – 32 bitový procesor – přidal k segmentům stránkování:
  - Souvislý LAP procesu není zobrazován jako jediná souvislá oblast FAP
- FAP se dělí na úseky zvané rámce
  - Pevná délka, zpravidla v celistvých mocninách 2
    - (512 až 8.192 B, nejčastěji 4KiB, ale někdy i 4 MiB)
- LAP se dělí na úseky zvané stránky
  - Pevná délka, shodná s délkou rámců
- Proces o délce  $n$  stránek se umístí do  $n$  rámců
  - rámce ale nemusí v paměti bezprostředně sousedit
- Mechanismus překladu logická adresa → fyzická adresa
  - pomocí tabulky stránek (PT = Page Table)
- Může vznikat vnitřní fragmentace
  - stránky nemusí být zcela zaplněny

# Stránkování – překlad adres

- Logická adresa použitá v programu se dělí na:
  - číslo stránky, p (index do tabulky stránek)
    - tabulka stránek obsahuje počáteční adresy rámců přidělených stránkám
  - posunutí (offset) ve stránce, d
    - relativní adresa (posunutí = offset, displacement) ve stránce/v rámci
- Protože velikost stránky je mocnina 2, je rozklad na číslo stránky a posunutí jednoduchý
  - číslo stránky  $p = \frac{addr}{2^k} = addr >> k$
  - posunutí  $off = addr \% 2^k = addr \& (2^k - 1)$
- V jazyce C je rozklad na číslo stránky pro 32-bitový systém s 4KiB stránkami
  - číslo stránky  $p = \text{addr} >> 12$
  - posunutí  $off = \text{addr} \& 0x00000FFF$

# Stránkování – bitová aritmentika

Velikost stránky - 4KiB, 12 bitů

```
#define PAGE_BITS 12
#define PAGE_SIZE (1 << PAGE_BITS)
// PAGE_SIZE 0x00001000
```

Maska pro zjištění pozice uvnitř stránky – offset

```
#define PAGE_MASK (PAGE_SIZE - 1)
// PAGE_MASK=0x00000FFF
```

získání rámce:

```
ramec = PT [addr >> 12] & (~PAGE_MASK)
RAM_addr = ramec | (addr & (PAGE_MASK))
```

Zaokrouhlení adresy na stránky dolů

```
static inline mword align_dn (mword val)
{
    val &= ~(PAGE_SIZE - 1);
// ~PAGE_MASK=0xFFFFF000
    return val;
}
```

Zaokrouhlení adresy na stránky nahoru

```
static inline mword align_up (mword val)
{
    val += (PAGE_SIZE - 1);
    return align_dn(val);
}
```

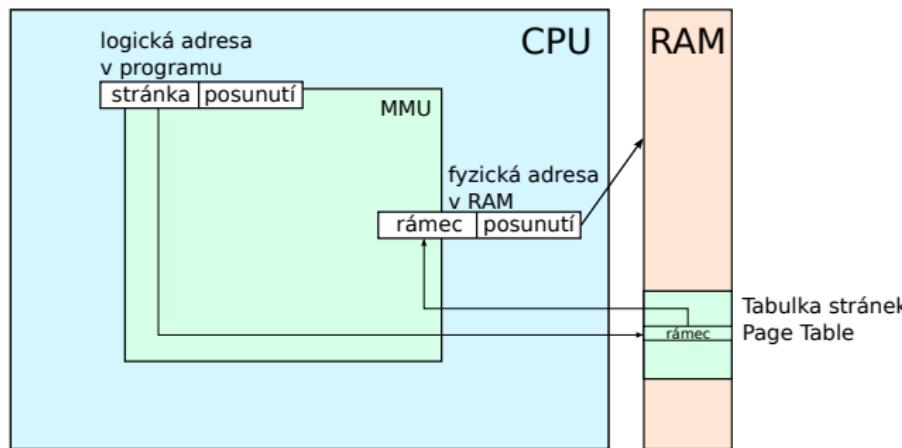
# Obsah tabulky stránek

Položky tabulky stránek:

- Číslo rámce
  - umístění stránky v reálné paměti počítače
- Atributy stránek
  - Základní příznaky
    - p** present – stránka je v paměti, číslo rámce je platné
    - ps** page size – velikost stránky
    - g** global – stránka je globální, nepatří jednomu procesu
  - Řízení přístupu
    - r/w** read/write – povolení zápisu do stránky
    - u/s** user/supervisor – povolení přístupu pro uživatele
  - Optimalizace
    - pwt** page-level write through – nastavení cache
    - pcd** page-level cache disable – zákaz použití cache
  - Statistika
    - a** accessed – stránka použita pro čtení
    - d** dirty – stránka použita pro zápis
  - Virtualizační příznaky
    - v/i** = valid/invalid indikuje přítomnost stránky ve FAP
    - a** = accessed značí, že stránka byla použita
    - d** dirty indikuje, že obsah stránky byl modifikován

# Efektivita tabulky stránek

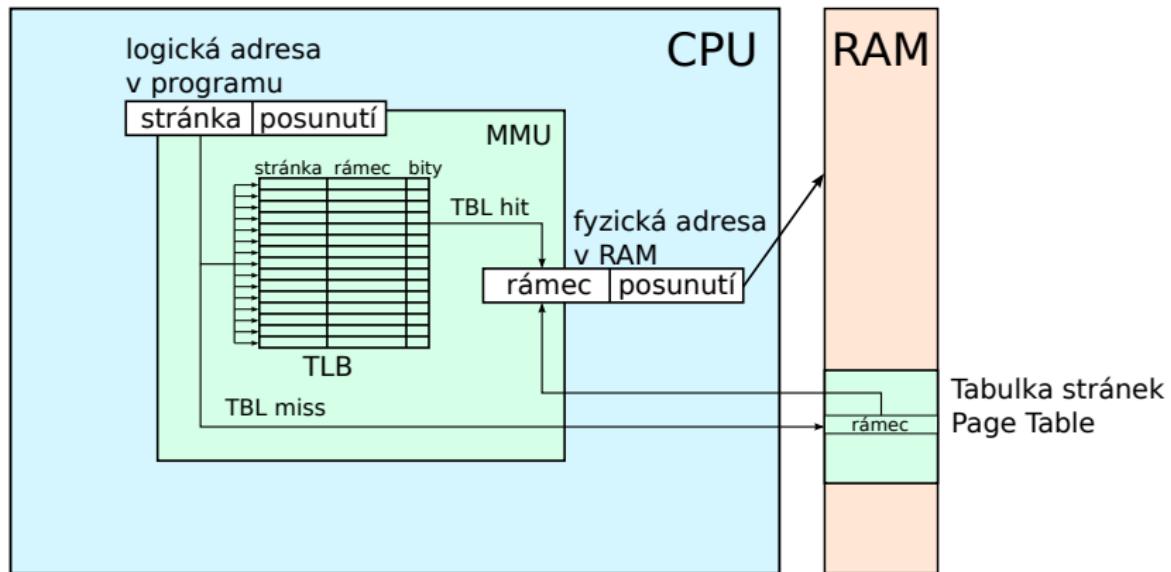
- každý přístup do paměti znamená, že je potřeba převést číslo stránky na číslo rámce, to znamená číst z RAM
- jedna instrukce může číst i dvakrát z paměti z různých stránek
- jedna instrukce tedy potřebuje 4 přístupy do paměti
- přístup do paměti velmi zdržuje



# TLB

- Řešení – speciální rychlá cache paměť pro čísla rámců a čísel stránek – Translation Lookaside Buffer
- asociativní paměť – paměť adresovaná obsahem
  - oproti normální paměti, ptám se, kde v paměti je hodnota 15?
- TLB je asociativní paměť
- relativně malá kapacita, vysoká efektivita a zrychlení přístupu do paměti
- nevýhoda – obvodová složitost implementace TLB
- MMU se zeptá TLB znáš hodnotu rámcce pro číslo stránky p?  
Odpověď buď ano je to  $r$  (TBL hit), nebo neznám (TBL miss)

# Tabulka stránek



# Význam TLB

- Skutečná přístupová doba – Effective Access Time (EAT) – 10–100 cyklů procesoru
- Přístupová doba TLB – 0.5 – 1 cyklus procesoru
- Neúspěšnost TLB, TLB miss – 0.01 – 1 %
- Příklad:
  - Přístupová doba do fyzické paměti  $t = 30\text{cyklu}$ , do TLB  $t_{TLB} = 1\text{cyklus}$
  - Neúspěšnost TLB  $\alpha = 0.01$  (jedno procento)
  - Stránkování bez TLB  $t_{celkem} = 2 \cdot t = 60\text{cyklu}$
  - Průměrná doba přístupu do paměti s TLB  
$$t_{celkem} = \alpha \cdot (t_{TLB} + 2 \cdot t) + (1 - \alpha) \cdot (t_{TLB} + t) = 31.3\text{cyklu}$$

## Vliv TLB

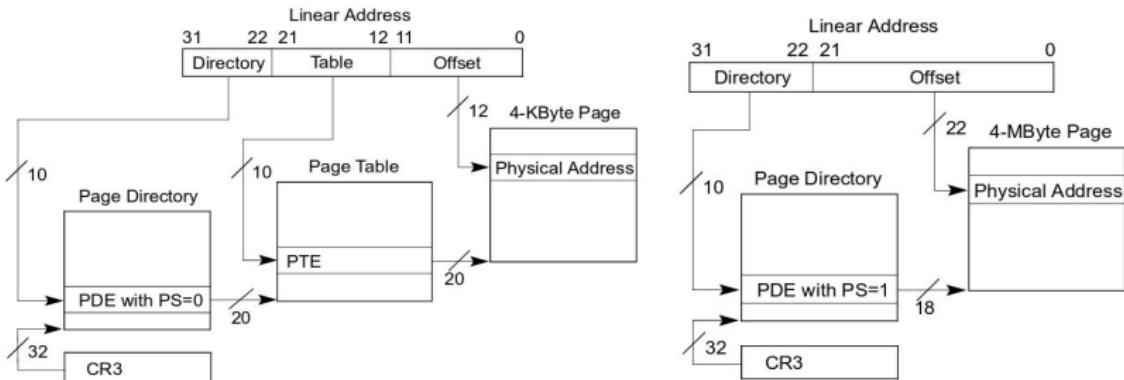
- Velikost TLB 8–4096 položek
- Moderní procesory mají více-úrovní TLB – podobně jako úrovně cache
- Intel Core i7 má 64 TLB položek L1 první úrovně a 1536 TLB položek L2 úrovně
- I pro malé TLB je úspěšnost nalezení položky 99–99.99% (souvisí s principem lokality tj. prostorovou závislostí programů)
- Problém TLB je při změně procesu a tím i změně tabulky stránek
- Intel umožňuje speciálními instrukcemi ponechat stránku v TLB

# Velikost tabulky stránek

- Otázka velikosti PT
  - Každý proces má svoji PT
  - 32-bitový LAP, 4 KiB stránky → PT má 1 Mi položek, tj. velikost 4 MiB
  - 64-bitový LAP, 4 KiB stránky – PT má 16 Pi (peta) položek, tj. velikost 128 PiB
  - musí být stále v hlavní paměti
- Hierarchické stránkování
  - Zobrazování LAP se dělí mezi více úrovní PT
  - Pro 32-bitový LAP typicky dvouúrovňové PT
  - PT 0 obsahuje definice (odkazy) vlastních tabulek PT 1
  - Tabulky stránek nižších úrovní mohou být odkládány na disk
  - v RAM lze zobrazovat jen skutečně využité stránky s vlastními PT
- Hašovaná PT
  - Náhrada přímého indexování číslem  $p$  v PT hašovací funkcí  $\text{hash}(p)$
- Invertovaná PT
  - Jediná PT pro všechny koexistující procesy
  - Počet položek je dán počtem fyzických rámců
  - Vyhledávání pomocí hašovací funkce  $\text{hash}(\text{pid}, p)$

# Dvouúrovňové stránkování 32-bitů

- 32-bitový procesor se stránkou o velikosti 4 KiB – 12 bitů posunutí (offset)
- 10 bitů index v tabulce tabulek (page directory –  $PT_0$ )
- 10 bitů index v tabulce stránek (page table –  $PT_1$ )
  - při nastaveném bitu PS – velikost stránky 4 MiB, nepoužije se tabulka  $PT_0$



# Více úrovňové stránkování (32bitů) – bitová aritmetika

Tabulka tabulek - vrchních 10 bitů adresy

```
pdir[virt >> 22]
```

Test přítomnosti tabulky stránek v paměti

```
if ((pdир[virt >> 22] & PRESENT) == 0)  
// tabulka stranek neni v pameti
```

Pozice tabulky stránek v paměti

```
ptab = pdир[virt >> 22] & ~PAGE_MASK;
```

Tabulka stránek - prostředních 10 bitů adresy

```
ptab[(virt >> PAGE_BITS) & 0x3ff]
```

# NOVA stránkování 32-bitů

```

#define PAGE_BITS      12
#define PAGE_SIZE      (1 << PAGE_BITS)
#define PAGE_MASK       (PAGE_SIZE - 1)

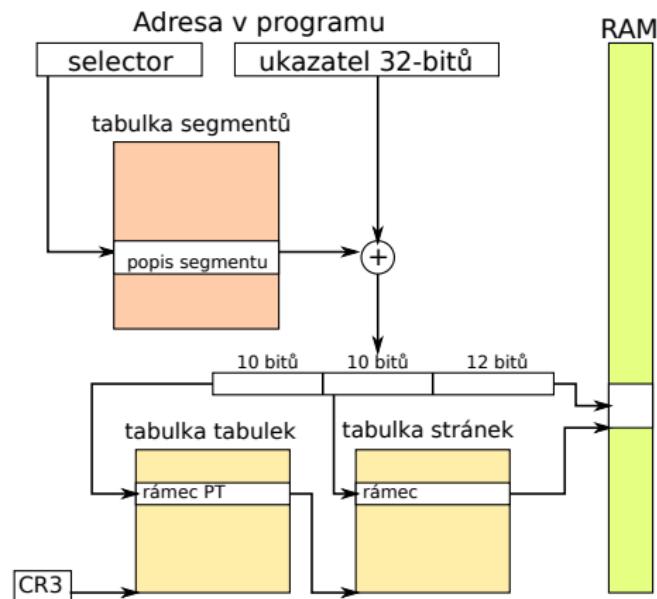
class Ptab {
public:
    enum {
        PRESENT = 1<<0,
        RW       = 1<<1,
        USER     = 1<<2,
        ACCESS   = 1<<5,
        DIRTY    = 1<<6, };
    static void insert_mapping (mword virt, mword phys, mword attr);
    static void * remap (mword addr);
};

void Ptab::insert_mapping (mword virt, mword phys, mword attr) {
    mword* pdir = static_cast<mword*>(Kalloc::phys2virt(Cpu::cr3()));
    mword* ptab;
    if ((pdir[virt >> 22] & PRESENT) == 0) { // add ptab
        ptab = static_cast<mword*>(Kalloc::allocator.alloc_page(1, Kalloc::FILL_0));
        mword p = Kalloc::virt2phys (ptab);
        pdir[virt >> 22] = p | ACCESS | RW | PRESENT | USER;
    } else { // find ptab
        ptab = static_cast<mword*>(Kalloc::phys2virt (pdir[virt >> 22] & ~PAGE_MASK));
    }
    ptab[(virt >> PAGE_BITS) & 0x3ff] = (phys & ~PAGE_MASK) | (attr & PAGE_MASK);
}

```

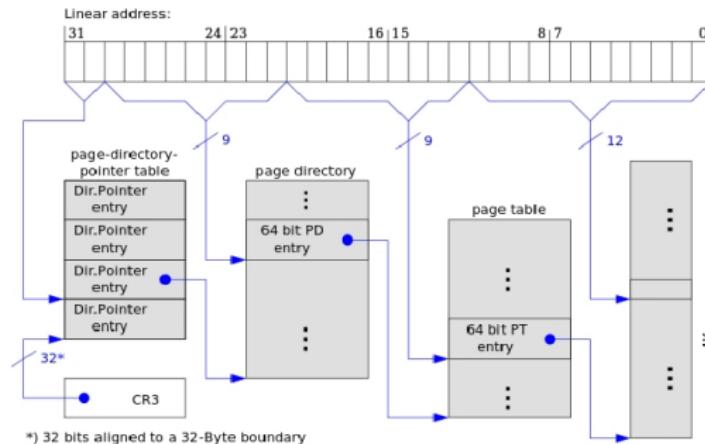
# Segmentace se stránkování IA-32

- V 32 bitovém módu nelze zrušit používání segmentů
- LAP: 2x8 Ki segmentů s délkou až 4 GiB každý
- Logická adresa = (popisovač segmentu, offset), offset = 32-bitová adresa v segmentu
- Lineární adresní prostor všech segmentů se stránkuje s použitím dvouúrovňového mechanismu stránkování
- Délka stránky 4 KiB, offset ve stránce 12 bitů, číslo stránky 2x10 bitů
- OS to řeší nafouknutím segmentů na 4GiB – což ve skutečnosti ruší segmenty



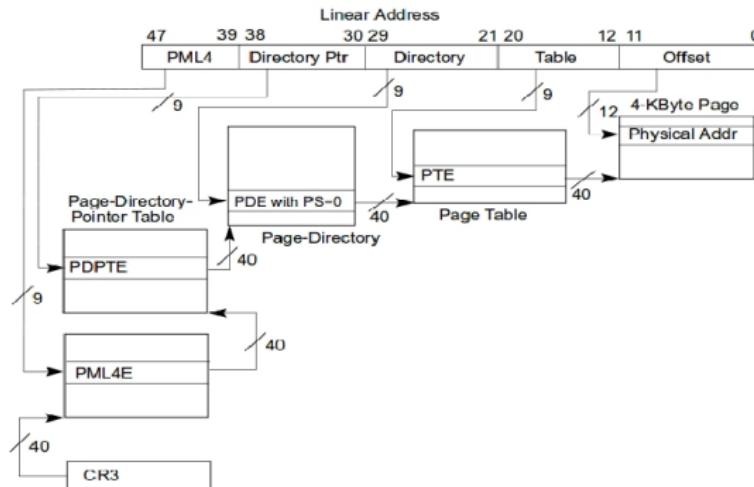
# Tříúrovňové stránkování 32-bitů s PAE

- PAE – fyzická paměť až 4 PiB – 52 bitů pro adresaci, tedy číslo rámce má 40 bitů
- potřebujeme více bitů pro uložení čísla rámce → položka v tabulce stránek bude mít 64 bitů
- do 4KiB se vejde jen 512 položek, tzn. index do této tabulky má 9 bitů
- 32-bitový procesor se stránkou o velikosti 4 KiB – 12 bitů posunutí (offset)
- 9 bitů index v tabulce tabulek
- 9 bitů index v tabulce stránek
- 2 bitů index v tabulce stránek druhé úrovně



# Stránkování IA-32e

- Lineární adresa 48 bitů – Virtuální prostor o velikosti 256 TiB
- Fyzická adresa 52 bitů – což je 4 PiB RAM
- Varianty s 4KiB, 2MiB nebo 1GiB stránkami
- Posunutí 12 bitů, 21 bitů, nebo 30 bitů
- 9 bitů indexy do tabulek tabulek/stránek



# Obsah

1 Správa paměti

2 Virtualizace paměti

# Virtualizace paměti

- Sdílený kód
  - Jediná read-only kopie kódu ve FAP sdílená více procesy
    - více virtuálních instancí editoru, shellů, jen jednou ve FAP.
- Privátní data
  - Každý proces si udržuje svoji vlastní virtuální kopii kódu a svoje reálná data
  - Stránky s privátním kódem a daty mohou být kdekoli v LAP
- Sdílená data
  - Potřebná pro implementaci mezičesní komunikace (mmap)

## Odkládání na disk

- Úsek FAP přidělený procesu je vyměňován mezi vnitřní a vnější (sekundární) pamětí oběma směry
  - Uložení stránek na disk, načtení z disku
  - Swap out, swap in (roll out, roll in)
  - Trvání výměn je z podstatné části tvořena dobou přenosu mezi pamětí a diskem je úměrná objemu vyměňované paměti
- Při nenalezení stránky ve fyzické paměti, nebo při porušení práv při přístupu do stránky nastane přerušení – chyba stránky (page fault)
- POZOR – některé části systému nelze odložit na disk
  - obsluha přerušení, která spravuje výpadek stránky
  - data potřebná k obsluze tohoto přerušení

# Principy stránkování

Kdy stránku zavádět do FAP? (Fetch policy)

- Stránkování při spuštění
  - Program je celý vložen do paměti při spuštění
  - velmi nákladné a zbytečné, předem nejsou známy nároky na paměť, dříve se nevyužívalo, dnes je využívána
- Stránkování či segmentace na žádost (Demand Paging/Segmentation)
  - Tzv. „líná metoda“, nedělá nic dopředu
  - Řeší problémy s dynamickou alokací proměnných
- Předstránkování (Prepaging)
  - Nahrává stránku, která bude pravděpodobně brzy použita
- Čištění (Pre-cleaning)
  - změněné rámce jsou ukládány na disk v době, kdy systém není vytížen
- Kopírovat při zápisu (copy-on-write)
  - Při tvorbě nového procesu není nutné kopírovat žádné stránky, ani kódové ani datové. U datových stránek se zruší povolení pro zápis.
  - Při modifikaci datové stránky nastane chyba, která vytvoří kopii stránky a umožní modifikace

# Líná metoda – Demand paging

- Při startu procesu zavede OS do FAP pouze tu část programu (LAP) kam se předává řízení – vstupní bod programu
  - Pak dochází k dynamickému zavádění částí LAP do FAP po stránkách „na žádost“ tj. až když je jejich obsah skutečně referencován
- Pro překlad LA → FA se využívá Tabulka stránek (PT)
  - Sada stránek procesu, které jsou ve FAP – rezidentní množina (resident set)
  - Odkaz mimo rezidentní množinu způsobuje přerušení výpadkem stránky (page fault) a tím vznikne „žádost“
    - Proces, jemuž chybí stránka, označí OS jako pozastavený
    - OS spustí I/O operace k zavedení chybějící stránky do FAP (možná bude muset napřed uvolnit některý rámec, viz politika nahrazování dále)
    - Během I/O přenosu běží jiné procesy; po zavedení stránky do paměti se aktualizuje tabulka stránek, „náš“ proces je označen jako připravený a počká si na CPU, aby mohl pokračovat
- Výhoda: Málo I/O operací, minimum fyzické paměti
- Nevýhoda: Na počátku běhu procesu se tak tvoří série výpadků stránek a proces se „pomalu rozbíhá“

# Princip lokality

- Odkazy na instrukce programu a data často tvoří "shluky"
- Vzniká časová lokalita a prostorová lokalita
  - Provádění programu je s výjimkou skoků a volání podprogramů sekvenční
  - Programy mají tendenci zůstávat po jistou dobu v rámci nejvýše několika procedur
  - Většina iterativních výpočtů představuje malý počet často opakovaných instrukcí,
  - Často zpracovávanou strukturou je pole dat nebo posloupnost záznamů, které se nacházejí v „sousedních“ paměťových lokacích
- Lze pouze dělat odhady o částech programu/dat, která budou potřebná v nejbližší budoucnosti

# Heuristiky stránkování

- Předstránkování (Pre-paging)
  - Sousední stránky LAP obvykle sousedí i na sekundární paměti, a tak je jich zavádění poměrně rychlé
    - bez velkých přejezdů diskových hlaviček
  - Platí princip časové lokality – proces bude pravděpodobně brzy odkazovat blízkou stránku v LAP. Zavádí se proto najednou více stránek
  - Výhodné zejména při inicializaci procesu – menší počet výpadků stránek
  - Nevýhoda: Mnohdy se zavádějí i nepotřebné stránky
- Čištění (Pre-cleaning)
  - Pokud má počítač volnou kapacitu na I/O operace, lze spustit proces kopírování změněných stránek na disk
  - Výhoda: uvolnění stránky je rychlé, pouze nahrání nové stránky
  - Nevýhoda: Může se jednat o zbytečnou práci, stránka se ještě může změnit

# Copy-on-write

## Kopírování až při zápisu

- velmi vhodné při vytvoření procesu – služba fork
- kód je sdílen, ten se nekopíruje ve FAP, pouze se připojí do nového virtuálního prostoru (zkopíruje se pouze odpovídající část stránkovací tabulky)
- data by měla být vlastní, měla by se vytvořit kopie dat ve FAP
  - není nutné to dělat, pokud nikdo (rodič ani potomek) nebude data měnit
  - to se dá pojistit zakázáním zápisu do stránek dat
    - při zápisu do stránky, nastane chyba stránkování (page fault) – OS zjistí, že je potřeba tuto stránku zkopirovat mezi rodičem a potomky (mohlo dojít k více voláním fork)
  - složitost této metody je dána možností vytvoření více potomků se stejnými datovými stránkami, z nichž některé jsou již zkopirovány a některé sdíleny

## Stránkování – politika nahrazování

- Co dělat, pokud není volný rámec ve FAP
  - Např. při startu nového procesu
- Politika nahrazování (Replacement Policy)
  - někdy též politika výběru oběti
- Musí se vyhledat vhodná stránka pro nahradu (tzv. oběť)
  - Kterou stránku „obětovat“ a „vyhodit“ z FAP?
  - Kritérium optimality algoritmu: minimalizace počtu (či frekvence) výpadků stránek

# Co dělá kdo?

## HW – CPU (MMU)

- 1 MMU automaticky převádí logickou adresu programu na fyzickou adresu podle tabulek stránek
- 2 když MMU nemůže převést logickou adresu na fyzickou vyvolá výjimku (přerušení)

## SW – operační systém

- 1 při svém zavádění nastaví CPU, aby používalo stránkování (tj. zapne MMU)
  - Kvůli zpětné kompatibilitě podporují moderní procesory více typů stránkování (32-bit, 64-bit, PAE, ...). OS si rozhodne, jaký typ se použije (většinou ten nejnovější, podporovaný jak HW, tak OS)
- 2 plní obsahy tabulek stránek, aby logické adresy odpovídaly určeným fyzickým rámcům (každý proces má vlastní tabulku stránek)
- 3 řeší výjimky (výpadky stránek) – přístup k virtuálním adresám, které:
  - nejsou mapovány do fyzické paměti (buď stránka na HDD (swap), nebo typicky chyby v programu, např. dereference NULL ukazatele)
  - jsou mapovány s jinými příznaky (např. zápis do read-only stránky ⇒ buďto chyba nebo aktivace copy-on-write)

# Stránkování – výběr oběti

- Určení oběti:
  - Politika nahrazování říká, jak řešit problémy typu:
    - Kolik rámců procesu přidělit?
    - Kde hledat oběti?
    - Jen mezi stránkami procesu, kterému stránka vypadla nebo lze vybrat oběť i mezi stránkami patřícími ostatním procesům?
- Některé stránky nelze obětovat
  - Některé stránky jsou dočasně „zamčené“, tj. neodložitelné
  - typicky V/V vyrovnávací paměti, řídicí struktury OS, ...
- Je-li to třeba, musí se rámec zapsat na disk („swap out“)
  - Nutné to je, pokud byla stránka od svého předchozího „swap in“ modifikována. K tomu účelu je v PT příznak dirty (modified) bit, který je automaticky (hardwarem) nastavován při zápisu do stránky (rámce).

# Algoritmus FIFO

Hledáme algoritmus, který je rychlý a vede na nejmenší počet výpadků stránek

- Obětí je vždy nejstarší stránka
- FIFO – jednoduché, rychlé, ale neefektivní
- Nevýhoda – i staré stránky se používají často

číslo rámce	1	2	3	4	1	2	5	1	2	3	4	5	3
1	1	1	1	1	1	1	5	5	5	5	4	4	4
2		2	2	2	2	2	2	1	1	1	1	5	5
3			3	3	3	3	3	3	2	2	2	2	2
4				4	4	4	4	4	4	3	3	3	3

Celkem 10 výpadků

# Optimální algoritmus

- Oběť – stránka, ke které bude přistupováno (čtení či zápis) ze všech nejpozději
  - tj. po nejdelší dobu se s ní nebude pracovat
- Budoucnost však v reálném případě neznáme
  - lze jen přibližně predikovat
- Lze užít jen jako porovnávací standard pro ostatní algoritmy
  - Zpětně při analýze jiných algoritmů „známe budoucnost“

číslo rámce	1	2	3	4	1	2	5	1	2	3	4	5	3
1	1	1	1	1	1	1	1	1	1	1	1	1	1
2		2	2	2	2	2	2	2	2	2	2	4	4
3			3	3	3	3	3	3	3	3	3	3	3
4				4	4	4	5	5	5	5	5	5	5

Celkem 6 výpadků

# Algoritmus LRU

- Predikce založená na historii
  - Předpoklad: Stránka, ke které nebylo dlouho přistupováno, nebude potřeba ani v blízké budoucnosti
- Oběť – stránka, ke které nejdéle nikdo nepřistoupil
  - LRU se považuje za nejlepší aproximaci optimálního algoritmu
  - bez všecké křišťálové koule lze těžko udělat něco lepšího

číslo rámce	1	2	3	4	1	2	5	1	2	3	4	5	3
1	1	1	1	1	1	1	1	1	1	1	1	5	5
2		2	2	2	2	2	2	2	2	2	2	2	2
3			3	3	3	3	5	5	5	5	4	4	4
4				4	4	4	4	4	4	3	3	3	3

Celkem 8 výpadků

# LRU – implementace

- Řízení časovými značkami
  - Ke každé stránce (rámcí) je hardwarově připojen jeden registr, do nějž se při přístupu do stránky hardwarově okopírují systémové hodiny (time stamp)
  - Při hledání oběti se použije stránka s nejstarším časovým údajem
  - Přesné, ale náročné jak hardwarově tak i softwarově
    - prohledávání časovacích registrů
    - každá instrukce musí modifikovat časovou značku 1–2 stránek
- Zásobníková implementace
  - Řešení obousměrně vázaným zásobníkem čísel referencovaných stránek
  - Při použití přesune číslo stránky na vrchol zásobníku
  - Při určování oběti se nemusí nic prohledávat, oběť je na dně zásobníku
  - Problém:
    - Přesun na vrchol zásobníku je velmi náročný, hardwarově složitý a nepružný; softwarové řešení nepřichází v úvahu kvůli rychlosti
    - Nutno dělat při každém přístupu do paměti!

# Aproximace LRU

- Příznak přístupu (Access bit, reference bit) – a-bit
  - Spojen s každou stránkou, po „swap-in“ = 0, při přístupu ke stránce hardwarově nastavován na 1
- Algoritmus druhá šance
  - Používá a-bit, FIFO seznam zavedených stránek – tzv. mechanismus hodinové ručičky
    - Každé použití stránky nastaví a-bit
    - Každé ukázání hodinové ručičky způsobí vynulování a-bitu (stránka dostane druhou šanci)
    - Obětí se stane stránka, na niž ukáže hodinová ručička a a-bit je nulový
  - Akce ručičky závisí na hodnotě a-bitu:
    - a=0: vezmi tuto stránku jako oběť
    - a=1: vynuluj a-bit, ponechej stránku v paměti a posuň ručičku o pozici dále
  - Jednoduché jako FIFO, při výběru oběti se vynescházá stránka aspoň jednou referencovaná od posledního výpadku
  - Numerické simulace – dobrá approximaci čistého LRU

# Algoritmus druhé šance

## ■ Ukázka příkladu s algoritmem druhé šance

číslo rámce	1	2	3	4	1	2	5
1	$1_{a=1}$	$1_{a=1}$	$1_{a=1}$	$1_{a=1}$	$1_{a=1}$	$1_{a=1}$	$5_{a=1}$
2		$2_{a=1}$	$2_{a=1}$	$2_{a=1}$	$2_{a=1}$	$2_{a=1}$	$2_{a=0}$
3			$3_{a=1}$	$3_{a=1}$	$3_{a=1}$	$3_{a=1}$	$3_{a=0}$
4				$4_{a=1}$	$4_{a=1}$	$4_{a=1}$	$4_{a=0}$

2	1	5	4	5	3
$5_{a=1}$	$5_{a=1}$	$5_{a=1}$	$5_{a=1}$	$5_{a=1}$	$5_{a=0}$
$2_{a=1}$	$2_{a=0}$	$2_{a=0}$	$2_{a=0}$	$2_{a=0}$	$3_{a=1}$
$3_{a=0}$	$1_{a=1}$	$1_{a=1}$	$1_{a=1}$	$1_{a=1}$	$1_{a=1}$
$4_{a=0}$	$4_{a=0}$	$4_{a=0}$	$4_{a=1}$	$4_{a=1}$	$4_{a=0}$

Celkem 7 výpadků

# Modifikovaná druhá šance

- Algoritmus označovaný též NRU (not recently used)
  - Vedle a-bitu se používá i bit modifikace obsahu stránky (dirty bit, d-bit)
    - nastavován hardwarem při zápisu do stránky
  - Hodinová ručička maže a-bity
    - proto je možná i stránka s nastaveným d-bitem a nulovým a-bitem

d	a	Význam
0	0	stránka se vůbec nepoužila
0	1	ze stránky se pouze četlo
1	0	stránka má modifikovaný obsah, ale dlouho se k ní nepřistupovalo
1	1	stránka má modifikovaný obsah a byla i nedávno použita

- Pořadí výběru (da): 00, 01, 10, 11
- Využití d-bitu šetří nutnost zápisu modifikované stránky na disk před odstraněním z paměti

# Přidělování rámců procesů

- Pevné přidělování
  - Procesu je přidělen pevný počet rámců
    - buď zcela fixně, nebo úměrně velikosti jeho LAP
  - Podhodnocení potřebného počtu rámců ⇒ velká frekvence výpadků
  - Nadhodnocení ⇒ snížení maximálního počtu spuštěných procesů
- Prioritní přidělování
  - Procesy s vyšší prioritou dostanou větší počet rámců, aby běžely „rychleji“
  - Dojde-li k výpadku, je přidělen rámec patřící procesu s nižší prioritou
- Proměnný počet rámců přidělovaných globálně (tj. z rámců dosud patřících libovolnému procesu)
  - Snadná a klasická implementace, užíváno mnoha OS (UNIXy)
  - Nebezpečí „výprasku“ (thrashing)
    - mnoho procesů s malým počtem přidělených rámců ⇒ mnoho výpadků
- Proměnný počet rámců přidělovaných lokálně (tj. z rámců patřících procesu, který způsobil výpadek)
  - Metoda tzv. pracovní množiny (working sets)

# Thrashing

- Jestliže proces nemá v paměti dost stránek, dochází k výpadkům stránek velmi často
  - nízké využití CPU
  - OS „má dojem“, že může zvýšit počet běžících vláken/procesů, aby se CPU více využilo, protože se stále se čeká na dokončení I/O operací
    - odkládání a zavádění stránek
  - Tak se dostávají do systému další procesy a situace se zhoršuje
- Thrashing – "Výprask" – počítač nedělá nic jiného než výměny stránek

# Pracovní množiny

Model pracovní množiny procesu  $P_i$  (working set)  $WS_i$

- Množina stránek, kterou proces referencoval při posledních  $n$  přístupech do paměti ( $n \sim 10.000$  – tzv. okno pracovní množiny)
- Pracovní množina je aproximace prostorové lokality procesu. Jak ji ale určovat?
  - Při každém přerušení od časovače lze např. sledovat a-bity stránek procesu, nulovat je a pamatovat si jejich předchozí hodnoty. Jestliže a-bit bude nastaven, byla stránka od posledního hodinového „tiku“ referencována a patří do  $WS_i$
  - Časově náročné, může interferovat s algoritmem volby oběti stránky, avšak účelné a často používané
  - Pokud suma všech  $WS_i$  (počítaná přes všechny procesy) převýší kapacitu dostupné fyzické paměti, vzniká „výprask“ (thrashing)
    - Snadná ochrana před „výpraskem“ – např. jeden proces se pozastaví

## Četnost výpadků stránek

- Linux nepočítá pracovní množiny, ale četnost výpadků stránek
- Pro každý proces se udržuje statistika, kolik výpadků stránek nastalo v čase
- Procesy s vyšší četností výpadků dostanou více reálné paměti
- Procesy v nižší četnosti mohou mít méně reálné paměti
- Thrashing nastane, pokud četnost výpadků všech procesů bude růst

# B4B350SY: Operační systémy

## Lekce 7. Alokace paměti

Petr Štěpán  
[stepan@fel.cvut.cz](mailto:stepan@fel.cvut.cz)



14. prosince, 2017

# Outline

- 1 Rozdělení paměti
- 2 Uživatelská alokace paměti
- 3 Alokace fyzické paměti

# Obsah

1 Rozdělení paměti

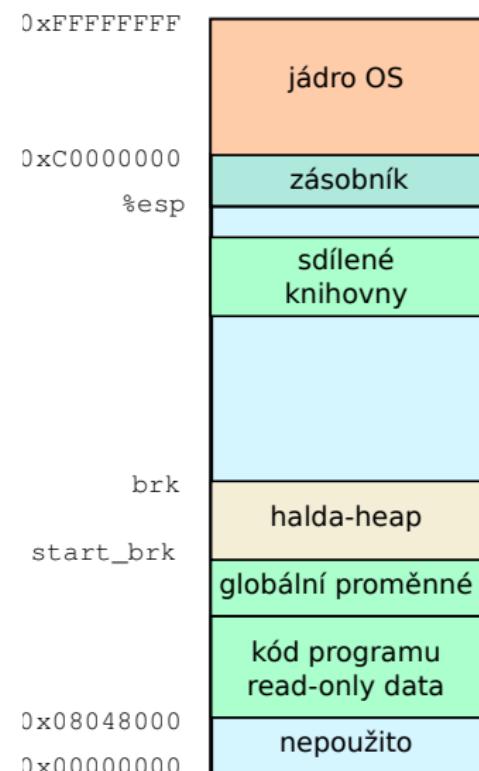
2 Uživatelská alokace paměti

3 Alokace fyzické paměti

# Rozdělení paměti 32-bitový systém Linux

- Virtuální paměťový prostor procesu je rozdělen na:

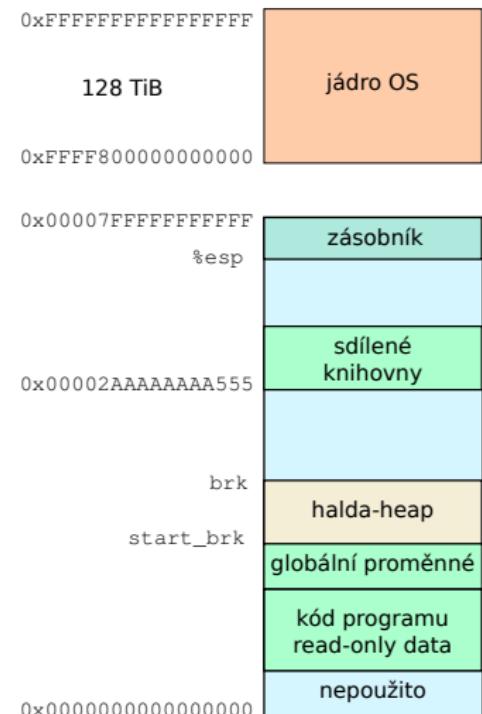
- systémovou část – dostupnou pro proces systémovými voláními (1GiB pro OS, Windows má dokonce 2GiB)
- uživatelskou část – prostor pro program, zásobník (pro všechna vlákna) a jeho data
  - část program, statická data
  - část halda – heap, dynamická data až do 3GiB
  - část mma – mapovaná paměť, dynamické knihovny
  - část zásobník, limit 8MiB



- paměťová mapa procesu je dostupná v /proc/\_pid\_/maps

# Rozdělení paměti 64-bitový systém

- Virtuální paměťový prostor procesu je rozdělen na:
  - systémovou část – dostupnou pro proces systémovými (128 TiB – virtuálně je místa dostatek)
  - uživatelskou část – prostor pro program, zásobník (pro všechna vlákna) a jeho data
    - část program, statická data
    - část halda – heap, dynamická data až do 42TiB
    - část mma – mapovaná paměť
    - část zásobník, limit 8MiB



# Obsah

1 Rozdělení paměti

2 Uživatelská alokace paměti

3 Alokace fyzické paměti

# Alokace paměti

- Paměť není neomezená
  - musí být alokována a spravována
  - mnoho aplikací velmi intenzivně potřebuje paměť
    - prohledávání stavového prostoru, chemická analýza složitých molekul, mapování v robotice
- Chyby při alokaci a správě jsou závažné a špatně detekovatelné
  - pokud je to možné, využívejte nástroje pro kontrolu alokace paměti
    - valgrind, -fsanitize=address
  - Přístup do paměti není vždy stejně rychlý
    - při programování je nutné brát ohled na využití cache, která je daleko rychlejší, než hlavní paměť počítače
    - úprava programu na lepší využití cache může významně zrychlit výpočet programu

# Alokace paměti

- Statická velikost, statická alokace
  - globální proměnné (i static proměnné jednotlivých modulů)
  - linker přiřadí místo pro globální proměnné ve virtuální paměti
  - zavaděč (OS) přiřadí konkrétní místo ve fyzické paměti
- Statická velikost, dynamická alokace
  - Lokální proměnné na zásobníku (stack)
  - překladač generuje kód tak, že k datům na zásobníku se přistupuje relativně vzhledem k jeho vrcholu (či tzv. frame pointeru)
- Dynamická velikost, dynamická alokace
  - kontroluje program
  - alokace na heapu – jak?

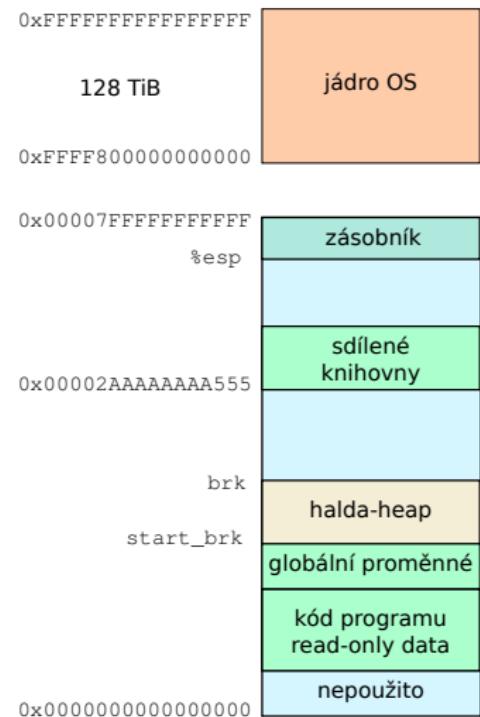
# Dynamická alokace

## ■ Explicitní a implicitní alokace

- Explicitní – program alokuje a uvolňuje paměť pro dynamické proměnné
  - např. funkce malloc a free v jazyce C, new/delete v C++
- Implicitní – program alokuje paměť pro nové proměnné, ale již je neuvolňuje
  - např. garbage collection v Javě nebo Pythonu
- Alokace v obou případech uvažuje paměť jako množina bloků

# Proces v paměti

- brk – omezení haldy, nejvyšší možná využitelná paměť (kromě zásobníku a sdílených knihoven)
- int brk(void \*addr) a void \*sbrk(intptr\_t increment)  
posouvají využitelnou paměť pro haldu
- zvětšení brk je spojené s alokací fyzické paměti a namapováním virtuální paměti na alokovanou fyzickou



# Balíček malloc

```
##include <stdlib.h>
```

- `void *malloc(size_t size)`

- Při úspěšné alokaci:

- vrací ukazatel na blok paměti o velikosti alespoň size bajtů, (typicky velikost zarovnaná na 8-bajtové bloky)
    - při size == 0, vrací NULL a nic nealokuje
    - Při nedostatku paměti: vrací NULL (0) a nastaví errno

- `void *calloc(size_t nmemb, size_t size)`

- Varianta malloc která navíc inicializuje paměť na 0

- `void free(void *ptr)`

- Vrací blok ptr do "bazénu" (pool) volných bloků paměti
  - ptr muselo být alokované původně funkcí malloc

- `void *realloc(void *ptr, size_t size)`

- Změní velikost bloku ptr a vrátí ukazatel na nový blok s novou velikostí
  - Obsah nového bloku se nezmění až do minima z hodnot nové a staré velikosti
  - Pokud nejde blok zvětšit, alokuje se nový blok a původní data se do něj zkopiují
  - Opět – ptr muselo být alokované původně funkcí (m/c)alloc, nebo realloc

# Cíle alokace

## ■ Hlavní cíle

- co nejrychlejší provedení funkcí malloc a free, měla by být rychlejší než lineárně k počtu alokovaných bloků
- minimalizovat fragmentaci paměti, co nejlepší využití paměti souvisí s minimální fragmentací (vnitřní i vnější)

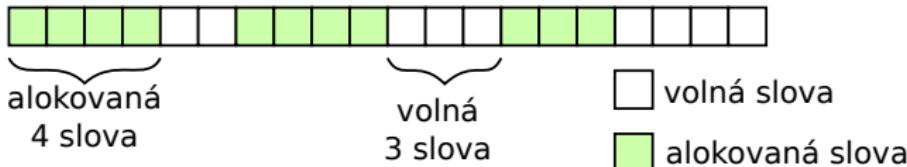
## ■ Vedlejší cíle

- Prostorová lokalita:
  - bloky alokované v podobném čase by měly být blízko u sebe
  - bloky podobné velikosti by měly být blízko u sebe
- Implementace by měla být robustní:
  - operace free by měla proběhnout pouze na správně alokovaném objektu
  - alokace by měla umožnit kontrolovat, zda se jedná o odkaz na alokované místo

# Předpoklady

## Konvence pro další část přednášky

- paměť je adresována po slovech (slovo v 32-bitových systémech je 4 bajtové, v 64-bitových systémech je 8 bajtové)
- čtverečky na obrázcích znamenají slovo
- každé slovo může obsahovat buď celé číslo, nebo ukazatel



# Problémy alokace

- Jak zjistit kolik paměti uvolnit při volání funkce free?
- Jak udržovat informaci o volných blocích v paměti?
- Co udělat s volným místem, pokud alokujeme paměť v díře, která je větší než požadované množství?
- Jak vybrat místo pro alokaci, když jich vyhovuje více?
- Jak vrátit alokovaný blok paměti po uvolnění do volných bloků?

# Velikost alokované paměti

Jak zjistit jak velká paměť se má uvolnit

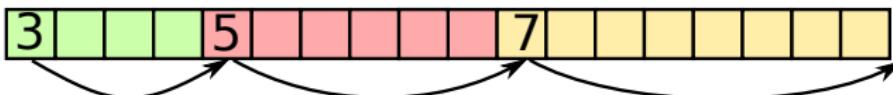
- Informaci o velikosti bloku lze uchovat před začátkem alokovaného bloku
- Číslo před začátkem udává velikost bloku a je označováno jako hlavička
- Informace vyžaduje dodatečné místo při alokaci



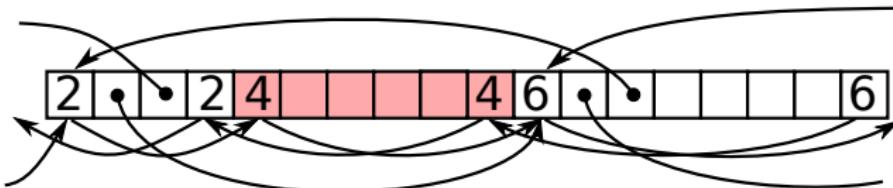
# Informace o volných blocích

4 základní metody udržování informace o alokovaných blocích a volném místě

- 1 implicitní seznam s použitím délky



- 2 explicitní seznam volných bloků

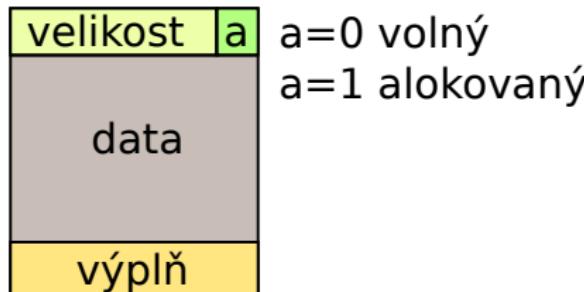


- 3 rozdílné seznamy volných bloků podle velikosti

- 4 uspořádaný seznam bloků podle velikosti

# Implicitní seznam

- potřebujeme identifikovat, zda je blok volný, nebo použitý
- rezervujeme extra bit
- bit použijeme ve stejném slově jako velikost bloku, protože velikost bloku v bajtech je vždy násobek slova (4 nebo 8 bajtů), můžeme použít nejnižší bit délky.



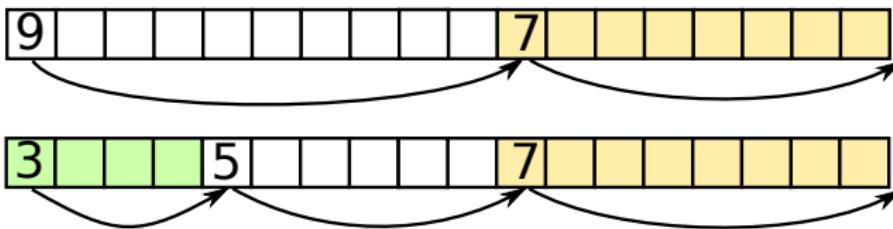
# Alokace paměti

- alokace znamená přepsání bitu "a" na 1, nebo rozdelení velkého bloku na dva
- který blok vzít pro alokaci:
  - první blok (first fit) – vezmi první blok od začátku haldy, který má alespoň požadovanou velikost
    - může být náročné, prohledávání celé haldy
    - nezohledňuje velikost, mohou vznikat mini-díry
  - další blok (next fit) – stejně jako první blok, ale pamatuje si kde skončil minule a začne na tom místě
    - rychlejší algoritmus, opět nezohledňuje velikosti
  - nejlepší blok (best fit) – najdi ze všech bloků ten, který má nejbližší velikost (nejlépe přesně požadovanou)
    - nejnáročnější, vždy prohledá celou haldu, vznikají mini-díry
  - nejhorší blok (worst fit) – najděte blok, který je největší
    - stačí si pamatovat, kde je největší blok, po rozdelení je nutné hledat největší blok

# Alokace paměti

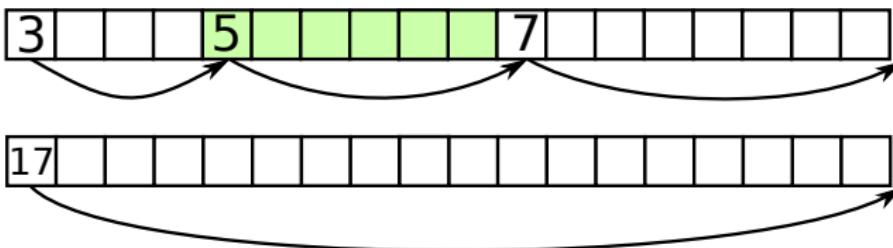
- blok volné paměti se při alokaci:

- rozdělí volný blok na dva bloky, jeden alokovaný, druhý volný
- celý volný blok se použije jako alokovaný, pokud by zbytek byl moc malý (nemohl by uchovat informaci o volném bloku)



# Uvolnění paměti

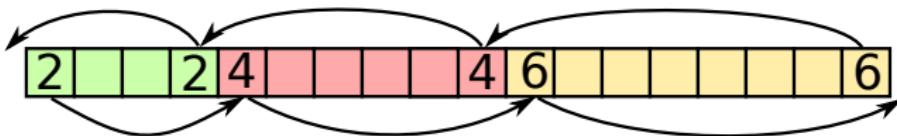
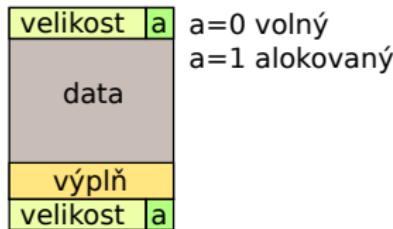
- uvolnění paměti je přepsání bitu "a" na 0 a spojení s předchozím a nebo následujícím volným blokem
- jak najít předcházející blok?
  - buď musíme projít celou haldou a spojovat dopředu
  - nebo musíme něco přidat do naší struktury bloku



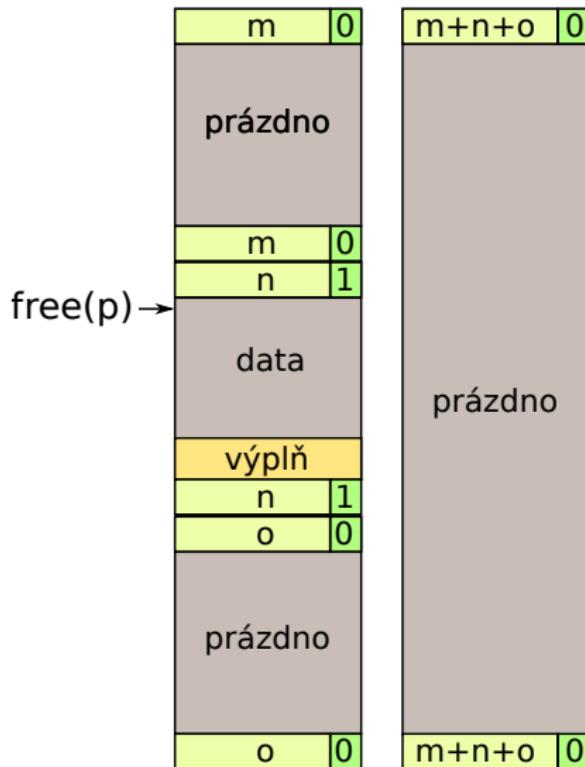
# Obousměrný implicitní seznam

Co přidat

- také na konec bloku přidáme jeho velikost [Knuth1973]
- spojení prázdných bloků v konstantním čase

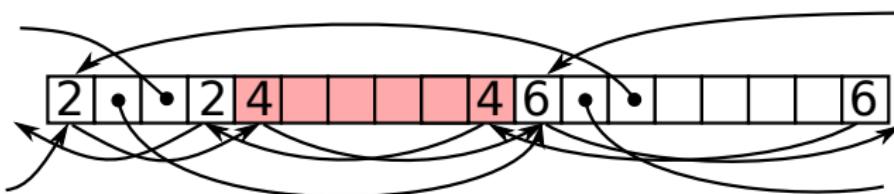


# Uvolnění paměti



# Explicitní seznam volných bloků

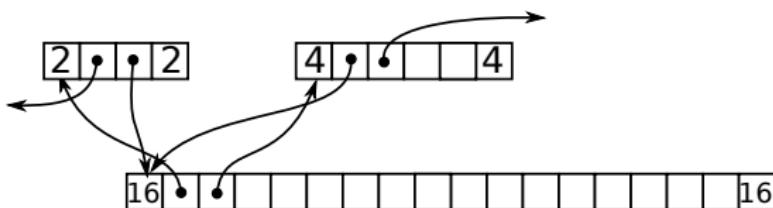
- princip využít volné místo v bloku pro uložení přímých ukazatelů na další a předchozí volný blok
- potřebujeme ponechat velikost bloku (na začátku i na konci), protože je nutná pro spojování uvolněných bloků



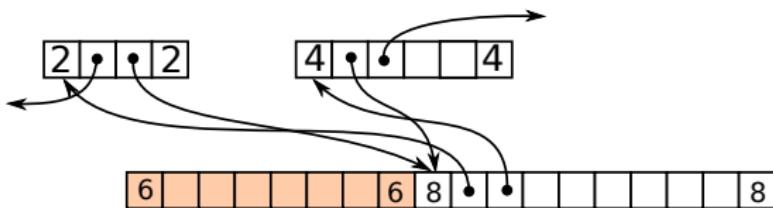
# Alokace s ukazateli

- pokud je využita jen část bloku, pak se jen posunou ukazatele

Před:



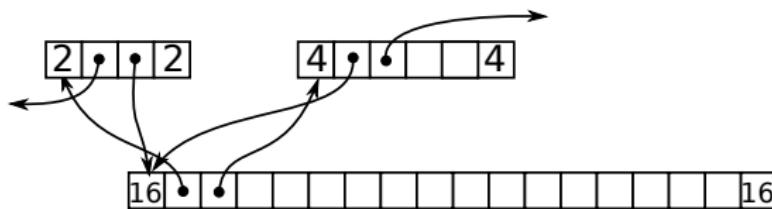
Po:



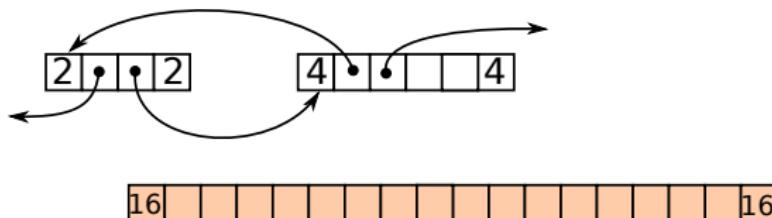
# Alokace s ukazateli

- pokud je využit celý blok, pak je třeba přepojit ukazatele předchozího a následujícího volného bloku

Před:



Po:



# Uvolnění bloku

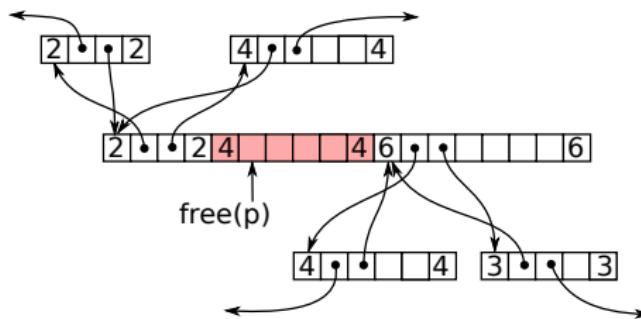
kam zařadit volný blok

- FIFO – zařadit blok na konec fronty (bere se i z prostřed, podle strategie výběru)
  - pro: rychlé, konstantní čase
  - proti: vyšší fragmentace
- podle adresy – udržovat setříděný seznam podle adresy
  - pro: studie ukazují, že je menší fragmentace
  - proti: zatřídění uvolňovaného bloku trvá déle

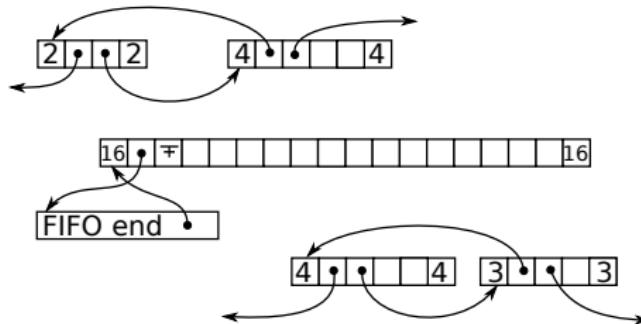
# Uvolnění bloku FIFO

- spojování bloků, nutné přepojit ukazatele původních bloků

Před:



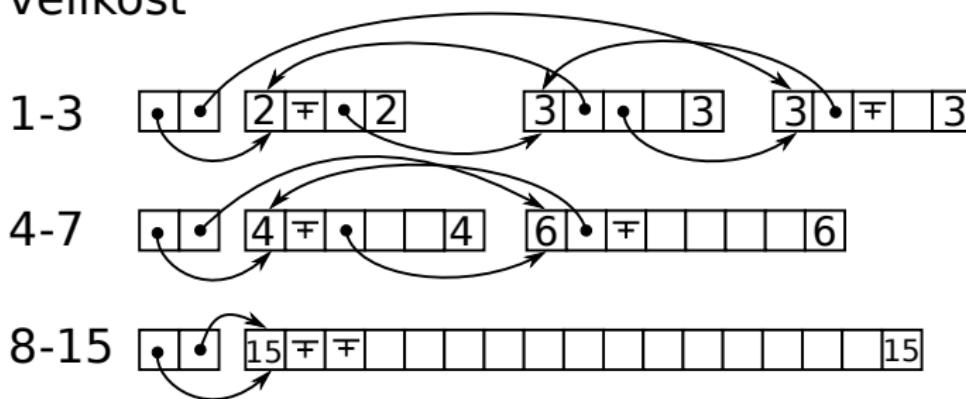
Po:



# Oddělené seznamy

- seznamy volných bloků podle jejich velikostí
- jeden seznam obsahuje bloky o velikosti v zadaném rozmezí
  - rozmezí většinou limity podle mocnin 2

velikost



# Oddělené seznamy

## ■ alokace bloku o velikosti $m$ :

- najdu první volný blok o velikosti  $n$ , že  $m < n$
- pokud je blok výrazně větší ( $n - m$  umožňuje začlenit blok do oddělených seznamů) pak vytvořím prázdný blok a vložím ho do seznamu volných bloků rozměru  $n - m$
- pokud blok není výrazně větší, pak použiji na alokaci celý nalezený blok

## ■ uvolnění bloku:

- je nutné zkontrolovat sousední bloky a pokud byly volné, vytvořit nový blok o větší velikosti a umístit ho do seznamu správné velikosti

# Vyhledávací struktura

- všechny volné bloky jsou setříděny ve struktuře, která umožní vyhledat nevhodnější velikost (best-fit) v čase –  $O(\log(n))$
- vyhledání prvního většího volného bloku je stejně náročné –  $O(\log(n))$
- nejčastěji se používají červeno-černé stromy (red-black tree), které jsou relativně jednoduché na implementaci a přitom efektivní

# Obsah

- 1 Rozdělení paměti
- 2 Uživatelská alokace paměti
- 3 Alokace fyzické paměti

# Zóny paměti x86

- NUMA (non-uniform memory access) – u víceprocesorových systémů trvá přístup do jednoho místa v paměti jinak dlouho, podle toho, z jakého procesoru přistupuji – souvisí s fyzickým umístěním paměťových čipů na základní desce
- UMA (uniform memory access) – jediná paměť se stejným přístupem
- Přestože PC jsou UMA i tak má paměť různé zóny, vzhledem k omezení přístupu periferií do fyzické paměti:

jméno	rozsah
ZONE_DMA	0–16 MiB of memory
ZONE_NORMAL	16–896 MiB
ZONE_HIGHMEM	896 MiB – End

- DMA – paměť vhodná pro použití komunikace s periferiemi, hlavně DMA přenosy HDD – RAM (starší periferie neuměly adresovat více než 16 MiB paměti)
- NORMAL – paměť celá mapovaná do oblasti jádra OS
- HIGHMEM – veškerý zbytek paměti, který se nevejde do NORMAL.

Pozor – 32bitový systém má pro jádro vyhrazen 1 GiB (někdy 2 GiB) adresového prostoru. Pokud je fyzické paměti více než 1 (2) GiB, nemůže být všechna fyzická paměť namapována do adresního prostoru jádra současně a mapování (obsah stránkovacích tabulek) se musí měnit podle toho, do jaké paměti je potřeba přistupovat. To hodně zpomaluje běh systému.

# Nano úvod do C++

```
class A {
public:
    enum B {Ex, Ey};
    int Ni;
    static int Si;
    A(int z): Ni(z) {}
    int f(int);
    static int Sf(int);
};
```

Znak `::` je použit pro definici a pro odkazy na statické prvky třídy A

```
int A::f(int x) {
    return -1*x;}
int A::Sf(int x) {
    return -2*x;}
// globální definice promenne
int A::Si;
```

Znak `::` je použit i při použití vnitřních struktur např. `enum B`

```
int m = A::Sf(A::Ex);
int n = A::Si;
```

Znak `.` je použit i pro přístup k prvkům instance třídy:

```
A a(10);
int m = a.f(A::Ey);
int n = a.Ni;
```

# kalloc.h a malé použití C++

## Třída Kalloc - zkráceno

```
class Kalloc {
private:
    const mword begin;
    mword end;
public:
    enum Fill {
        NOFILL = 0, FILL_0,
        FILL_1};

    static Kalloc allocator;
    Kalloc (mword virt_begin,
            mword virt_end):
        begin (virt_begin), end (virt_end) {}
    void * alloc_page (unsigned size,
                       Fill fill = NOFILL);
    void free_page (void *);

    static void * phys2virt (mword);
    static mword virt2phys (void *);
};
```

## Třída Kalloc - definice

```
void * Kalloc::phys2virt (mword phys) {
    mword virt = phys +
        reinterpret_cast<mword>(&OFFSET);
    return reinterpret_cast<void*>(virt);
}

mword Kalloc::virt2phys (void * virt) {
    mword phys = reinterpret_cast<mword>(virt)
        - reinterpret_cast<mword>(&OFFSET);
    return phys;
}

Třída Kalloc - použití

void *stack = Kalloc::allocator.alloc_page(1,
    Kalloc::FILL_0);
if (!Ptab::insert_mapping (0x1000,
    Kalloc::virt2phys (stack),
    Ptab::PRESENT | Ptab::RW | Ptab::USER)) {
    panic ("Not enough memory\n");
}
```

# Přechod virtualní a fyzická adresa

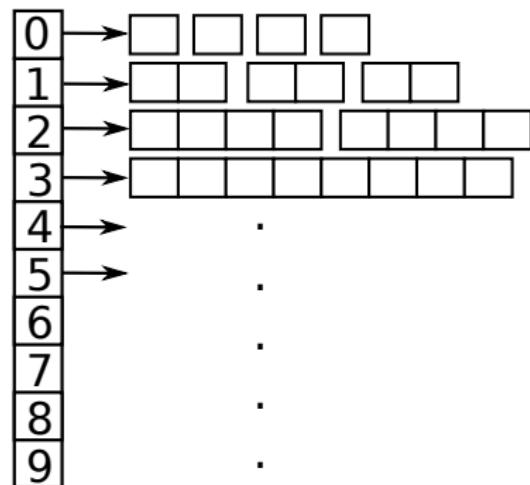
Jak je možné, že přechod mezi virtuální a fyzickou adresou je tak jednoduchý?

```
void * Kalloc::phys2virt (mword phys) {  
    mword virt = phys + reinterpret_cast<mword>(&OFFSET);  
    return reinterpret_cast<void*>(virt);  
}
```

- Platí to vždy?
- Ne - platí to jen pro úsek paměti, který je celý namapován do fyzické paměti, jako jeden blok
- DŮLEŽITÉ: Po zapnutí stránkování nemůže ani JOS přistoupit přímo k fyzické paměti
  - Celé jádro OS se pohybuje také ve virtuálním prostoru
- OS si pro sebe zabere zónu DMA a NORMAL a využívá ji přednostně pro alokaci objektů, u kterých potřebuje znát fyzickou adresu
  - Tabulky stránek
    - OS musí vyplnit rámec tabulky stránek do tabulky tabulek
  - Přístup k souborům na disku přes DMA
    - OS musí vyplnit rámec, kam se budou kopírovat data z DMA
    - DMA neví nic o stránkování
    - DMA není procesem, ale HW

# Zóny paměti

- každá zóna si udržuje seznam volných a použitých rámců
- pokud počet volných rámců klesne pod stanovenou mez, spouští se swap démon, který začne připravovat odložení stránek na disk
- při dosažení spodní limitní hranice se alokující proces blokuje do ukončení uvolňování stránek
  - některé procesy nelze blokovat, ty mohou provést alokace i pod tento spodní limit
- každá zóna si uržuje seznam volného místa v blocích stránek



# Zóny paměti

- seznam řádu **k** znamená, že udržuje volné bloky o velikosti  $2^k \times PAGE\_SIZE$
- pokud není volný rámec požadovaného řádu, vezme se nejbližší volný blok vyššího řádu, rozdělí se poloviny a přesune do nižšího řádu, případně rekurzivně až do požadovaného řádu
- uvolnění bloku může vést ke spojení se sousedním blokem a přechod do vyššího řádu
- k detekci možného spojení slouží bitová mapa pro všechny bloky daného řádu
- každá zóna má seznamy řádu 0 až MAX (většinou 10 – blok 4MiB)
- zóna má funkce `alloc_pages`, `free_pages` pro alokaci bloku stránek zadанého řádu a jeho uvolnění

# Mapa rámců paměti RAM

- každý rámec fyzické paměti má v paměti alokovanou strukturu, která popisuje jeho využití

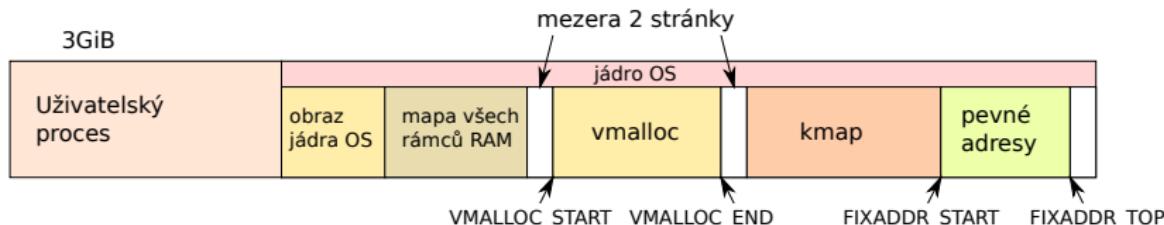
```
typedef struct page {  
    struct list_head list;  
    struct address_space *mapping;  
    unsigned long index;  
    struct page *next_hash;  
    atomic_t count;  
    unsigned long flags;  
    struct list_head lru;  
    struct page **pprev_hash;  
    struct buffer_head * buffers;  
#if defined(CONFIG_HIGHMEM) || defined(WANT_PAGE_VIRTUAL)  
    void *virtual;  
#endif /* CONFIG_HIGMEM || WANT_PAGE_VIRTUAL */  
} mem_map_t;
```

# Rámce paměti

- uvedená struktura slouží pro uchování všech informací o rámci:
  - list – odkaz na spojový seznam rámců (např. clean, dirty, lock)
  - mapping – odkaz na soubor, který tento rámec mapuje do paměti
  - count – kolik stránek odkazuje na tento rámec (např. sdílené stránky kódu procesů)
  - flags – příznaky stránky (např. active, inactive, unused, dirty, slab, lru)
  - lru – odkaz do spojového seznamu pro algoritmus lru – active/inactive seznam
  - buffer – odkaz hlavičky bufferů, pokud stránka slouží jako disková cache, nebo má svoji kopii ve swap
  - virtual – (volitelně) odkaz na virtuální adresu stránky, pro HIGMEM rámce v jádru (má pouze jednu virtuální adresu)

# Rozložení paměti jádra (Linux)

- obraz jádra – rozbalený kód jádra OS
- mapa rámců paměti – struct page pro každý rámec RAM
- oblast vmalloc – oblast pro vmalloc
- oblast kmap – oblast kam jsou dočasně mapovány stránky z HIGHMEM
- oblast fixovaných adres – adresy, které je potřeba znát před spuštěním jádra např. ACPI



# Alokace v jádře

- kmalloc – funkce pro alokaci dynamických objektů v prostoru jádra, vytváří fyzicky i virtuálně spojité úseky paměti. Pro alokaci používá algoritmus známý jako SLAB.
- vmalloc – alokace celých stránek (spojitých ve virtuální paměti, ne nutně spojitých ve fyzické paměti)
- GFP (get free page) příznaky – kde, co a jak alokovat
  - GFP\_ATOMIC – nelze proces uspat, je nutné dokončit alokaci přímo
  - GFP\_NOHIGHIO – alokace v jádře, proces může být uspán, ale požaduje paměť ze zóny NORMAL
  - GFP\_KERNEL – obyčejná alokace v jádře, proces může být uspán a vzbuzen po uvolnění paměti, zóna HIGHMEM
  - GFP\_HIGHUSER – obyčejná alokace normálního uživatele v zóně HIGHMEM, může být uspán

## vmalloc

- alokuje v jádře blok paměti, který ve fyzické paměti nemusí být souvislý
- kroky vmalloc:
  - najdi místo ve virtuální paměti pro daný blok
  - alokuj potřebný počet stránek pro daný blok
  - upraví referenční tabulkou jádra OS
  - při prvním přístupu procesu k alokovanému prostoru se vyskytne chyba stránky a OS nakopíruje informace o rámci z referenční tabulky stránek OS do tabulky stránek OS.
- velikost bloků paměti alokovaná vmalloc je zaokrouhlena na stránky
- mezi jednotlivé alokované bloky je vložena prázdná stránka, která chrání před přesahy (off-by-one chyby) mezi bloky paměti

# kmalloc

- SLAB (SLUB) algoritmus
- SLAB má tři základní cíle:
  - alokovat malé bloky paměti a při tom zabránit fragmentaci paměti
  - udržovat volné nejčastěji používané objekty pro jejich rychlé opětovné využití (cache uvolněných objektů stejné velikosti, např. datagramy)
  - lepší využití procesorové L1 a L2 cache zarovnáním objektů na velikost L1, nebo L2 cache
- informace o využití SLAB cat /proc/slabinfo
- objekty jsou alokované z tzv. cache, tj. struktur, které obsahují alokované stránky pro objekty dané velikosti
  - standardně se vytvoří cache pro základní typy velikostí (obdoba oddělených seznamů podle velikostí)
  - cache vytváří uživatelé, tedy části jádra OS, pokud chtějí zrychlit alokaci mnoha malých kousků paměti
  - kmalloc při požadavku na alokaci paměti najde cache, která nejlépe odpovídá požadované velikosti
  - cache zajistí alokaci zjištěním volného místa v slab – kontejneru na několik objektů zadанé velikosti

# Cache pro kmalloc

- každá cache má 3 seznamy slab podle obsazenosti:
  - plný – tento slab nelze použít pro alokaci
  - poloprázdný – tento slab je kandidátem pro alokaci nového objektu
  - prázdný – tento slab je kandidátem pro uvolnění
- každá cache se snaží udržovat rozumný počet volného místa pro další alokaci objektů zadané velikosti
- od jádra 2.6 se vytváří také cache per-CPU
  - tyto cache jsou spojeny s procesory a snaží se, aby objekty, které patří jednomu vláknu byly umístěny ve stejné L1, nebo L2 cache, tedy fyzicky blízko
  - tato procesorově orientovaná alokace umožňuje, aby se data při běhu lépe vešla do procesorové cache a tím aby proces běžel rychleji

# B4B350SY: Operační systémy

## Lekce 8: Bezpečnost (security)

Michal Sojka  
[michal.sojka@cvut.cz](mailto:michal.sojka@cvut.cz)



November 12, 2020

# Osnova

## 1 Základní koncepty kybernetické bezpečnosti

- Co je to bezpečnost?
- Cíle zabezpečení
- Zabezpečení OS
- Mechanismy a politiky
- Další pojmy, příklady

## 2 Řízení přístupu

## 3 Stack overflow

- Co je přetečení zásobníku?
- Útoky a ochrany proti nim

# Cíl přednášky

- Vysvětlit vám základní koncepty používané v kybernetické bezpečnosti
- Předat znalosti potřebné pro vyřešení (nepovinné) úlohy – útok pomocí přetečení zásobníku

# Co je to bezpečnost?

- Pro každého něco jiného



I stalk



# Počítačová (kybernetická) bezpečnost

## Information security

- Ochrana **mých zájmů** (počítačem ovlivnitelných) před nepřátelskými hrozbami
- Velmi individuální a subjektivní
  - Různí lidé mají různé zájmy
  - Různí lidé čelí různým hrozbám
- Neexistují univerzální řešení
  - Je počítač s Windows dostačující k uložení přísně tajných informací?
    - Je připojen k internetu? Kdo k němu má přístup – fyzicky i vzdáleně?
  - Tvrzení, že systém je bezpečný má smysl jen vzhledem k dobře definovaným **cílům zabezpečení**, které definují
    - **hrozby** (tj. proti čemu jsme systém zabezpečili – např. napadení známým virem)
    - **bezpečné stavy** systému (tj. za jakých podmínek je systém považován za bezpečný – např. antivirus a firewall jsou zapnuty)

# Cíle zabezpečení

- Definují **co** má být chráněno, **proti komu** a za jakých **podmínek**.
- Před nasazením každého systému se nad tím aspoň zamyslete!
  - Často je potřeba to mít sepsané a schválené (od zákazníka, šéfa, ...).
- **Co** má být chráněno se často vyjadřuje pomocí tzv **CIA vlastností**:
  - **Důvěrnost** (**Confidentiality**)
    - X se nesmí dozvědět o Y
    - Příklad: Franta se nesmí dozvědět administrátorské heslo.
  - **Integrita**
    - X nesmí narušit Y
    - Příklad: Karel nesmí měnit libovolné záznamy v databázi uživatelů.
  - **Dostupnost** (**Availability**)
    - X nesmí způsobit nedostupnost Y pro Z
    - Příklad: Žádný uživatel nesmí způsobit pád web serveru.

# Typické prostředky pro zajištění CIA vlastností

	What to protect?		
	Confidentiality	Integrity	Availability
Perspectives			
Cryptography	Encryption	Digital Signatures Hashing/MAC	
Network Security	TLS VPN Routing, Firewalling		Replication
Operational Security	Password Strength Audit Training, Best Practices, Certified Processes Reproducible Builds		
Posthumous Security	Malware Analysis Honeypots Forensics, Debugging Log Anomaly Analysis		
Reactive Security	Antivirus Taint Tracking Intrusion Detection	Lockstepping Log Monitoring	Watchdog
Attack Mitigation	ASLR Pledges Stack-Smash Protection Pointer Obfuscation		
Resilience	Formal Verification Compartmentalization, Sandboxing, Virtualization	Redundancy	Secure Boot Self-Healing

Source: <https://genodians.org/nfeske/2019-07-11-security>

# Současný stav zabezpečení OS

## ■ Historicky:

- Zaostával za vývojem potřeb uživatelů
  - Např.: Bezpečnostní řešení byla zaměřena na firemní („enterprise“) zákazníky
- Zaostával za objevujícími se hrozbami
  - Zaměření na ochranu uživatelů mezi sebou (práva k souborům na disku), ne na ochranu uživatelů před nedůvěryhodnými aplikacemi

## ■ V některých ohledech se **zlepšuje**:

- Např. OS chytrých telefonů používají důkladnější zabezpečení než desktopy
- Objevuje se méně kritických bezpečnostních dér v běžných OS

## ■ V jiných se **zhoršuje**:

- Velikost, funkcionalita a složitost OS stále roste
- Jen málo lidí skutečně ví, jak psát bezpečný kód
- Stále více lidí umí na systémy útočit

# Bezpečnost operačních systémů

- Co by mělo být cílem OS v oblasti bezpečnosti?
- Minimálně:
  - poskytovat **mechanismy** umožňující tvorbu bezpečných systémů,
  - které jsou být schopny bezpečně implementovat uživatelem či administrátorem nastavenou **politiku**
  - a to tak, aby tyto mechanismy nebylo možné obejít.
- Bezpečnost systému je tak silná, jak silný je **nej slabší článek**.
  - Ďábel je skryt v detailech
  - ...i proto vás učíme assemblér :-)

# Dobré mechanismy zabezpečení

- Jsou široce použitelné
- Podporují obecné principy bezpečnosti (viz příští slide)
- Je snadné je správně a bezpečně použít
- Nejsou v rozporu s jinými (nebezpečnostními) prioritami – např. s produktivitou práce.
- Dají se snadno správně implementovat i verifikovat

# Běžné mechanismy zabezpečení v OS

- **Systémy pro kontrolu přístupu** – kontrola, k čemu může daný proces přistupovat (např. práva k souborům)
- **Autentizační systémy** – potvrzení identity toho, jehož „jménem“ proces běží (login name, heslo, ...)
- **Logování** – kvůli auditům, detekci útoků, vyšetřování a obnovu
- **Šifrování souborových systémů** – HW lze šifrovat celý disk, SW jen souborový systém (nelze šifrovat partition table)
- **Správa pověření (credentials)**
- **Automatické aktualizace**

Bezpečnost je „prorostlá“ celým systémem. Neexistuje jen jedna komponenta zodpovědná za bezpečnost.

# Rozdíl mezi politikou a mechanismem

- Politiky doprovázejí mechanismy
  - Politika **řízení přístupu** (access control)
    - K čemu mají přístup sekretářky a k čemu vývojáři?
  - Politika **autentizace**
    - Je heslo o 5 znacích dostatečné pro přístup do KOSu?
- Politika často omezuje použitelné mechanismy
  - Nestačí heslo, je potřeba se prokázat certifikátem
- Co někdo považuje za politiku, ostatní za mechanismus :-)

# Zásady bezpečnostního designu

Saltzer & Schroeder [SOSP '73, CACM '74]

- **Úspornost mechanismů** – keep it stupid simple (KISS)
- **Bezpečné výchozí nastavení** – nikomu se nechce trávit čas nastavováním (vždy dobrá inženýrská praxe)
- **Kompletní zprostředkování** – kompletní kontrola dat předávaných z „venku“ do vnitřních částí systému (ničemu, co jde „z venku“ nevěřit)
  - SQL injection, Cross-site Scripting (XSS), ...
- **Otevřený návrh** – ne „security by obscurity“
- **Oddělení pravomocí** – možnost dál uživatelům jen ta práva která potřebují (ne, jen běžný uživatel, který nemůže „nic“ a root/admin, který může vše)
- **Nejmenší pravomoci** – uživatelé nemají žádná oprávnění, která nepotřebují
- **Co nejméně společných mechanismů** – minimalizace sdílení dat, minimalizace závislostí (knihoven, balíků)
- **Psychologická přijatelnost** – pokud se to těžce používá, nikdo to používat nebude
  - např. ve starších Windows měl většinou každý uživatel přiřazeny administrátorská práva, protože nebyl jednoduchý způsob, jak tato práva získat jen pro jednu operaci. Škodlivý program je tím pádem mohl automaticky zneužít.

# Jeep hack

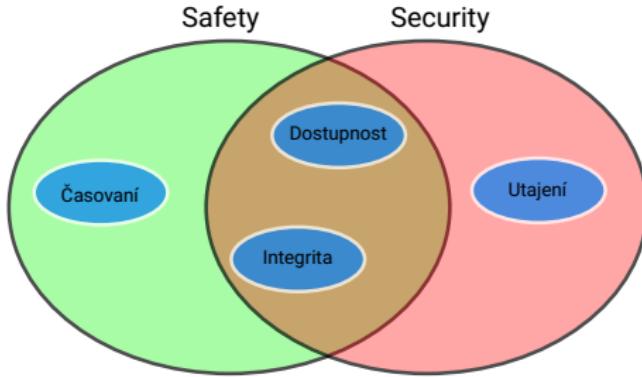
Miler & Valasek, 2015



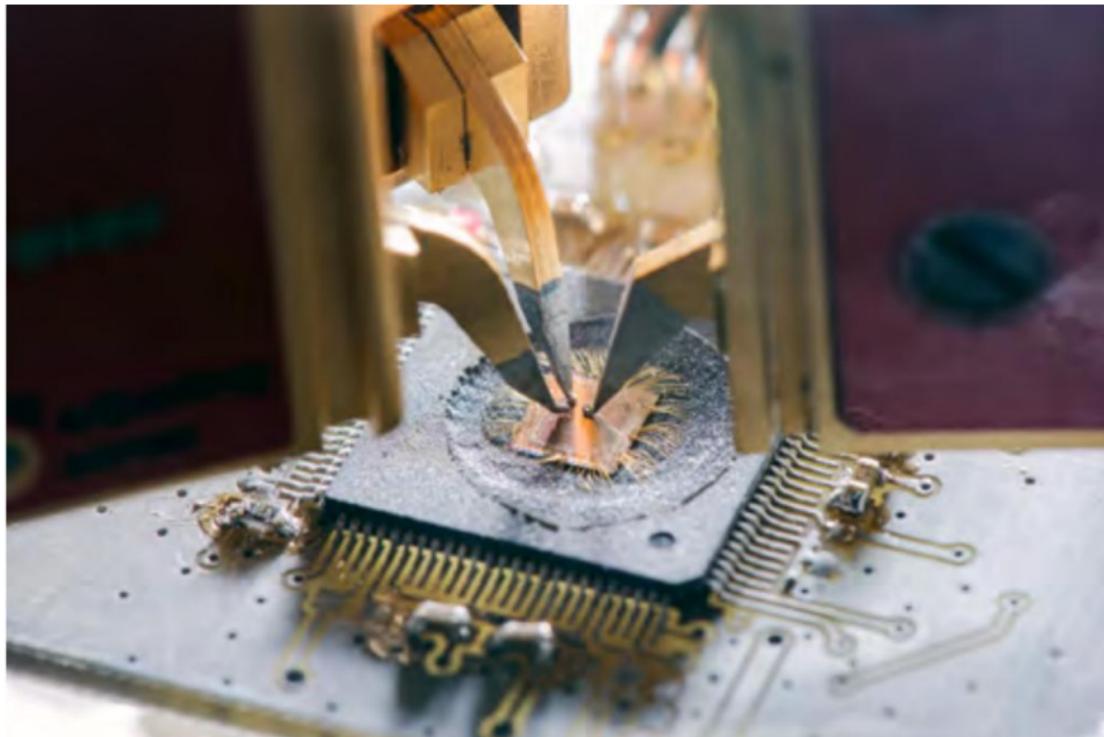
- Služba v „infotainment“ jednotce byla přes D-Bus omylem dostupná z internetu
- Útočníci nahráli SSH klíč, spustili SSH server, přeprogramovali řadič sběrnice CAN, aby šel ovládat přes sériovou linku z infotainment jednotky, ...
- <https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/>

# Více druhů bezpečnosti

- Angličtina rozlišuje dva termíny „safety“ a „security“, které se do češtiny (i jiných jazyků) překládají oba jako bezpečnost.
- **Safety** – ochrana okolí před systémem
  - letadlo nezpůsobí škody v okolí (smrt lidí, škody na majetku)
- **Security** – ochrana systému před okolím
  - hacker neovládne vaše auto
  - teroristé nezpůsobí pád letadla, srážku vlaků
- Vztah cílů zabezpečení k safety a security (výjimky existují):



## Útoky jsou sofistikované



Využití postranních kanálů – zde se hledají šifrovací klíče pomocí měření elektromagnetického pole v okolí čipu

# Důvěra & Trusted Computing Base

- Všechny systémy obsahují entity, kterým se **věří**
  - pokud selžou, systém nemusí být bezpečný
  - hardware, OS, administrátor serveru, ...
- **Trusted Computing Base (TCB):**
  - množina všech takových entit
  - Co je součást TCB při práci s internetovým bankovnictvím?
- Bezpečné systémy musí mít **důvěryhodné TCB**
  - minimalizace TCB je klíčem k důvěryhodnosti.

# Souhrn

- Bezpečnost je velmi subjektivní, jsou potřeba dobře definované cíle zabezpečení
- Bezpečný OS by měl poskytovat:
  - dobré bezpečnostní **mechanismy**
  - podporující různé uživatelské **politiky**.
- Bezpečnost je daná **důvěryhodností** klíčových entit
  - **TCB**: množina všech klíčových entit
  - OS je nezbytně součástí TCB

# Mechanismy a politiky pro řízení přístupu

Podrobnější pohled na jeden z aspektů zabezpečení OS.

## ■ Politika

- Specifikuje, kdo má povolen přístup k čemu
- a jak se to může měnit v čase

## ■ Mechanismus

- Implementuje politiky (viz dále)

## ■ Některé mechanismy nabádají k některým politikám

- Některé politiky nejdou vyjádřit pomocí mechanismů ve vašem OS

# Základní princip

**Matice řízení přístupu** [Lampson'71] definuje **stav ochrany** v daném čase

- Objekty jsou např. soubory
- Subjekt je např. uživatel
- Subjekty mohou být zároveň objekty

	Obj1	Obj2	Obj3	Subj2
Subj1	R	RW		send
Subj2		RX		control
Subj3	RW		RWX	recv

# Ukládání stavu ochrany

- Typicky ne jako matice
  - moc „řídké“, neefektivní, dynamické
- Dvě zřejmé volby:
  - 1 Ukládání jednotlivých sloupců dohromady s objektem
    - Každý sloupec je nazýván „seznam pro řízení přístupu“ (**access control list**, ACL) daného objektu
  - 2 Ukládání jednotlivých řádků dohromady se subjektem
    - Definuje objekty, ke kterým má daný subjekt přístup – doména ochrany (**protection domain**) daného subjektu
    - Každý takový řádek je nazýván „seznam schopnosti“ (**capability list**)

# Seznamy pro řízení přístupu (ACL)

- Implementováno skoro ve všech běžných OS
- Subjekty jsou obvykle sloučeny do tříd
  - např. v UNIXu: majitel, skupina, ostatní

```
$ ls -ld /var/spool/cups
drwx--x--- 1 root lp 6754 Nov 22 00:00 /var/spool/cups
```
  - obecnější seznamy ve Windows či v současném Linuxu (příkazy get/setfacl)
  - mohou obsahovat „negativní“ oprávnění – např. pro vyloučení přístupu několika uživatelů ze skupiny
- Meta-oprávnění
  - řízení členství ve třídách
  - dovolují modifikaci ACL

Obj1	
Subj1	R
Subj2	
Subj3	RW

# Schopnosti (capabilities)

- **Schopnost** [Dennis & Van Horn, 1966] je prvek seznamu schopností

Subj1	Obj1	Obj2	Obj3	Subj2
	R	RW		send

- Pojmenovává objekt, aby s ním program mohl zacházet (Obj2)
- Uděluje k objektu práva (RW)
- Všichni, kdo vlastní „schopnost“ mají právo s objektem pracovat
- Použití
  - Méně časté v komerčních systémech
  - KeyKOS (VISA transaction processing) [Bomberger et al, 1992]
  - Capsicum capabilities (FreeBSD)
  - Častěji ve výzkumných OS: NOVA, EROS, L4Re kernel
    - Mnohé z těchto systémů se nyní začínají prosazovat i v komerční oblasti

# ACL vs. schopnosti

## Seznamy řízení přístupu (ACL)

- Proces musí být schopen **zjistit jaké objekty existují** (pojmenování) a pak teprve je může používat (a nebo mu je k nim přístup odepřen)
- Typicky to řeší tzv. **ambientní autorita** – tj. každý proces má všechna práva uživatele, který ho spustil (např. „vidí“ celý souborový systém a může zjistit, kteří další uživatelé jsou v systému.
  - Pokud program spouští jiný program, potomkovi nelze jednoduše práva omezit.
  - V Linuxu se dnes tento problém řeší pomocí „jmenných prostorů“ (**namespaces**), ale není to elegantní a trpí to některými nedostatkami

## Schopnosti

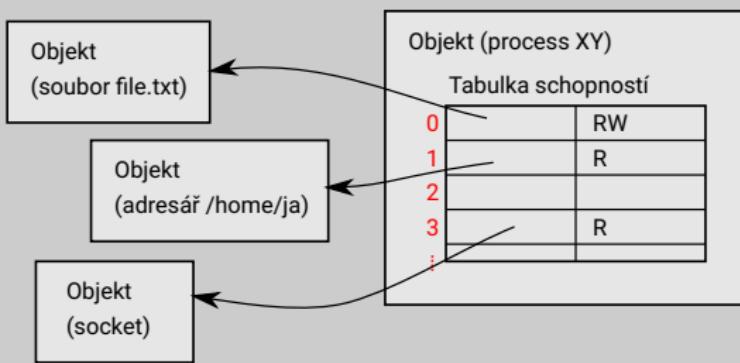
- Neexistuje ambientní autorita, každému procesu jsou **delegovány** jen ty **schopnosti**, které potřebuje (zásada nejmenší pravomoci).
- Např. proces nevidí všechny soubory, ale jen soubory (či celé adresářové stromy), které mu rodič nebo nějaká služba „delegoval(a)“.
- Nikdo nemůže delegovat schopnosti, které sám nemá.

# Možná implementace a použití schopností

Uživatelský proces XY

```
write(0, "Hello");
int program;
while ((program = readdir(1)) != -1) {
    if (getname(program) == "myprog.exe") {
        // spusť /home/ja/myprog.exe a deleguj mu objekt socket na indexu 2
        child = create_process(program, capabilities=[-1, -1, 3]);
    }
    revoke(program);
}
```

Jádro OS



- Schopnosti jsou podobné jako „file descriptors“ v UNIXových OS.
  - Program identifikuje schopnosti číslem.
  - Schopnosti jdou získat jen tak, že nám ne někdo deleguje.
- Např.:

- Rodič při vytváření potomka
- Služba souborového systému jako odpověď na požadavek otevření souboru.

# Povinné vs. nezávazné řízení přístupu

Mandatory vs. Discretionary Access Control

Bezpečnostní mechanismy pro řízení přístupu se dají rozdělit do dvou skupin:

- Nezávazné řízení přístupu (DAC):
  - Uživatelé mohou sami rozhodovat o přístupu (např. měnit práva ostatních uživatelů ke svým souborům)
  - Mohou delegovat svá přístupová práva ostatním uživatelům
- Povinné řízení přístupu (MAC)
  - Je vynucována administrátorem definovaná politika
  - Uživatelé ji nemohou měnit (pokud to politika explicitně nepovoluje)
  - Může zabránit nedůvěryhodným aplikacím běžícím s právy uživatele v páchnání škody.

# Přetečení bufferu (buffer overflow)

- Jedna z nejčastějších chyb programátorů v C, často i v C++
- Skoro vždy se dá nějak zneužít
- Hodně zajímavé (z hlediska útočníků) je přetečení bufferu na zásobníku (lokální proměnná)
  - Tzv. stack smashing attack
  - Zneužitelnost chyby je dnes na velkých systémech (servery, PC) částečně eliminována (viz dále)
  - Problém je ale...

# IoT

## Internet of Things Insecurity

# Zásobník – jak tam jsou uložena data?

Záleží, jestli je kód přeložen s nebo bez „frame pointeru“

```
int func(int arg1, int arg2)
{
    volatile int val=arg1+arg2;
    return val;
}

void main()
{
    func(1, 3);
}
```

Zásobník uvnitř func()

	arg2 = 3
ebp+0x8	arg1 = 1
	návratová adresa
ebp	uložený base pointer
ebp-0x4	lokální proměnná val
	(ostatní lokální prom.)
esp	

S ukazatelem rámce (gcc -fno-omit-frame-pointer)

func:

```
push %ebp ; ulož bp na zásobník
mov %esp,%ebp ; nastav ebp jako na obr.
sub $0x10,%esp ; nastav esp jako na obr.
mov 0xc(%ebp),%eax ; načti arg2
add 0x8(%ebp),%eax ; přičti k arg1
mov %eax,-0x4(%ebp) ; ulož do prom val
mov -0x4(%ebp),%eax ; zkopíruj val do eax
leave ; obnov ebp
ret ; vrať se do main
```

main:

```
push %ebp
mov %esp,%ebp
push $0x3 ; ulož parametry funkce
push $0x1 ; ... na zásobník
call 500 <func> ; zavolej func
pop %eax
pop %edx
leave
ret
```

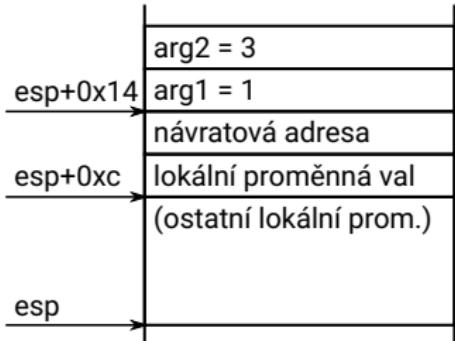
# Zásobník – jak tam jsou uložena data?

Záleží, jestli je kód přeložen s nebo bez „frame pointeru“

```
int func(int arg1, int arg2)
{
    volatile int val=arg1+arg2;
    return val;
}

void main()
{
    func(1, 3);
}
```

Zásobník uvnitř func()



Bez ukazatele rámce (gcc -fomit-frame-pointer)

```
func:
    sub    $0x10,%esp      ; nastav esp jako na obr.
    mov    0x18(%esp),%eax ; načti arg2
    add    0x14(%esp),%eax ; přičti k arg1
    mov    %eax,0xc(%esp)   ; ulož do prom. val
    mov    0xc(%esp),%eax   ; zkopíruj val do eax
    add    $0x10,%esp      ; posuň esp k návr. adr.
    ret
```

; vrát se do main

```
main:
    push   $0x3            ; ulož parametry funkce
    push   $0x1            ; ... na zásobník
    call   500 <func>       ; zavolej func
    pop    %eax
    pop    %edx
    ret
```

; ... na zásobník

; zavolej func

# Přetečení zásobníku

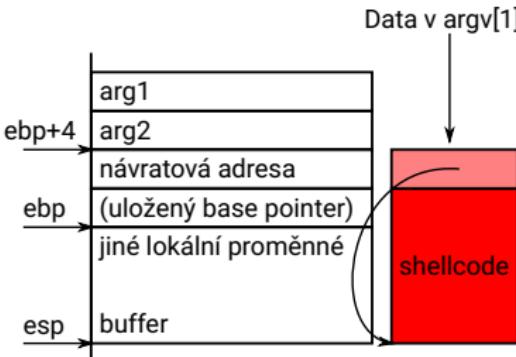
- Programátor zapomene zkontrolovat velikost proměnných na zásobníku
- Uživatel může předat programu víc dat, než je velikost proměnné na zásobníku
- To může způsobit přepsání dalších lokálních proměnných, návratové adresy, parametrů, ...
- Program pak většinou „spadne“ (segmentation fault)
- Nebo toho můžeme zneužít a donutit program, aby dělal to, co chceme my.

# Zneužití přetečení zásobníku

Chceme, aby cizí program dělal to, co chceme my a ne to, co chtěl jeho autor.

```
int main(int argc, char *argv[])
{
    char buffer[10];
    strcpy(buffer, argv[1]);
    return 0;
}
```

Spuštění: ./prog "\$(cat shellcode)"



## Shellcode

- Typickým cílem útočníka je spuštění shellu, tj. chtěl by spustit následující kód:
 

```
dup2(socket, 0);
dup2(socket, 1);
execve("/bin/sh", NULL, NULL);
```
- Strojový kód většinou nesmí obsahovat binární nuly, protože funkce jako `strcpy` předpokládají řetězec ukončený nulou a nezpracovaly by všechna data
- Příklad: Instrukce `mov \$1,%eax` ukládá do `eax` hodnotu 1. Ve svém kódování má 3 nulové byty, protože 32bitová hodnota 1 tj. 0x00000001 je součást instrukce. Tuto instrukci můžeme nahradit vynulováním instrukcí `xor` a zvětšením o jedna:

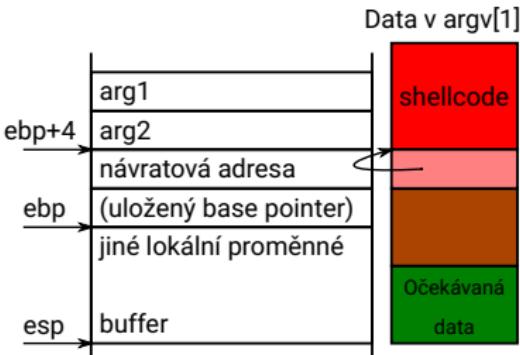
```
B8 01000000 mov $1,%eax
                  // nahradit za
33C0             xor %eax,%eax
40               inc %eax
```

# Zneužití přetečení zásobníku

Chceme, aby cizí program dělal to, co chceme my a ne to, co chtěl jeho autor.

```
int main(int argc, char *argv[])
{
    char buffer[10];
    strcpy(buffer, argv[1]);
    return 0;
}
```

Spuštění: ./prog "\$(cat shellcode)"



## Shellcode

- Typickým cílem útočníka je spuštění shellu, tj. chtěl by spustit následující kód:  
`dup2(socket, 0);  
dup2(socket, 1);  
execve("/bin/sh", NULL, NULL);`
- Strojový kód většinou nesmí obsahovat binární nuly, protože funkce jako strcpy předpokládají řetězec ukončený nulou a nezpracovaly by všechna data
- Příklad: Instrukce `mov $1,%eax` ukládá do eax hodnotu 1. Ve svém kódování má 3 nulové byty, protože 32bitová hodnota 1 tj. 0x00000001 je součást instrukce. Tuto instrukci můžeme nahradit vynulováním instrukcí xor a zvětšením o jedna:  

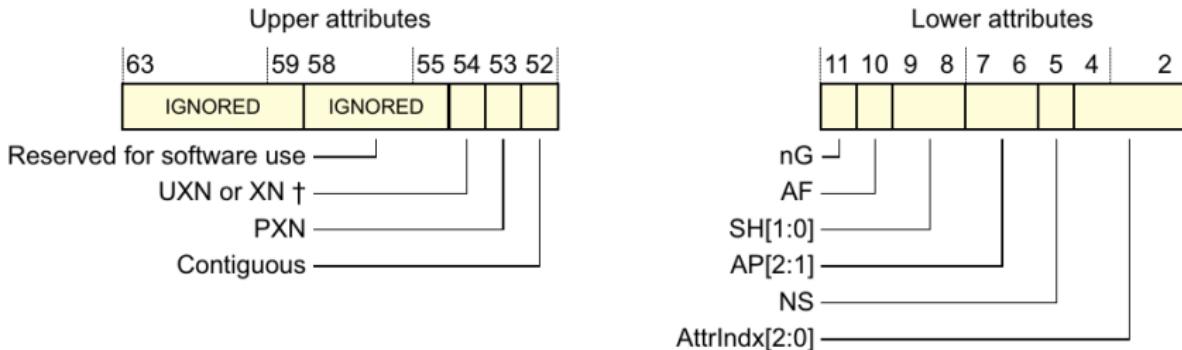
B8 01000000	mov \$1,%eax
33C0	// nahradit za
40	xor %eax,%eax
	inc %eax

# Nespuštiteľný zásobník

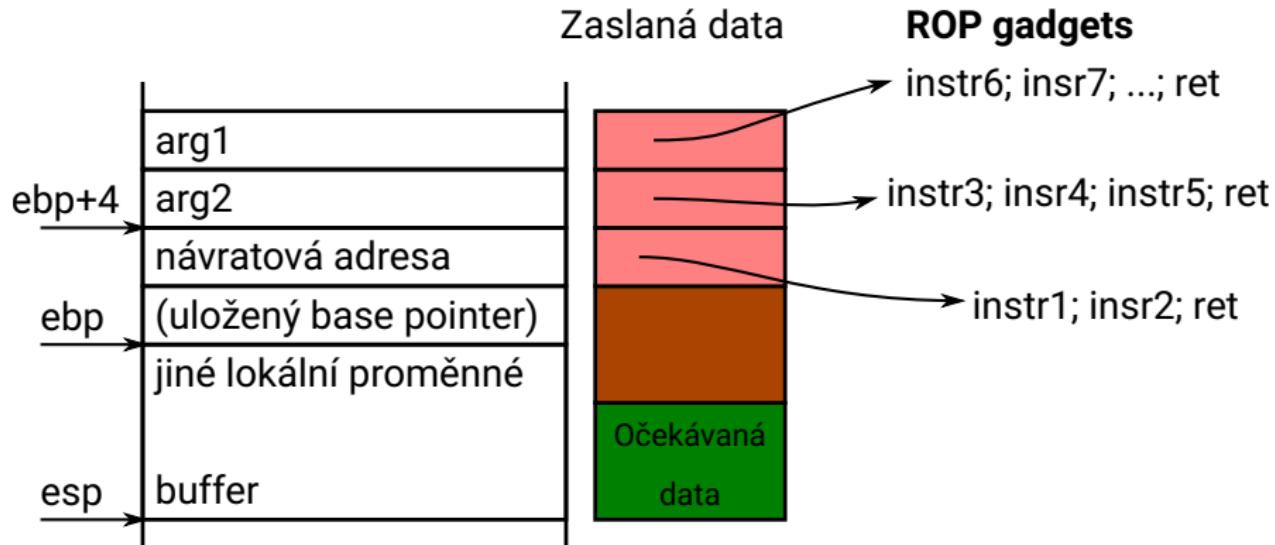
- Intel zavedl od PAE stránkování (PAE a x86\_64) XD bit (eXecute-Disable)
  - Při pokusu o vykonání kódu ze stránky s XD=1 dojde k výjimce.
  - Paměť pro zásobník se alokuje s XD=0.

X D	Reserved	Address of 4KB page frame	Ign.	P G	A A	D D	P C W	P U S	R /S W	1	PTE: 4KB page
--------	----------	---------------------------	------	--------	--------	--------	-------------	-------------	--------------	---	---------------------

- ARM používá UXN/XN bit (Unprivileged eXecute Never)



# Return-oriented programming (ROP)



- Když nejde spustit (shell)kód na zásobníku, může se útočník pokusit spustit kód, který už v programu je.
- Že by se v programu nacházel přesně ten kód, který útočník potřebuje, je nepravděpodobné.
- Jinde je ale spousta „zajímavého“ kódu – např. v knihovně `libc` najdeme kód, který vyvolává všechna možná systémová volání.
- ROP = Útočník neposílá přímo shellcode, ale sekvenci návratových adres, které způsobí postupné vykonání kousků kódu (ROP gadgets), který se nachází v jiných částech programu.

# ROP – pokračování

- Existují ROP překladače
  - Předloží se jimi program a knihovny, na které chceme útočit (např. webový server a knihovny z populární Linuxové distribuce)
  - kompilátor přeloží zdrojový kód v C do ROP programu (sekvence návratových adres, které je potřeba uložit na zásobník).

# Náhodné rozložení adresního prostoru

Address space layout randomization (ASLR)

- Pro většinu typů útoků se zásobníkem je potřeba znát adresy, na které lze „skákat“ instrukcí ret.
- Pokud útočník neumí adresy zjistit, jsou útoky těžké či nemožné.
- Sdílené knihovny jsou zkompilovány tak, že je lze nahrát a spustit z libovolné adresy (position independent code – PIC)
  - Linkování se provádí až při spuštění, takže je možné je umístit při každém startu na jinou adresu.
- I program lze přeložit jako PIC (značí se PIE – position independent executable) a zásobník také nemusí být na pevné adrese.
- Zkuste si v GNU/Linuxu spustit: `watch -d cat /proc/self/maps`. Uvidíte, že při každém spuštění příkazu `cat` jsou adresy jiné.

# Když ASLR nestačí

- Jádra OS nemohou používat tak intenzivní ASLR jako uživatelský prostor.
- Linux používá náhodnou adresu zásobníků v jádře, ale adresa kódu se zvolí náhodně jen při bootu, pak zůstává stejná.
- Možná řešení: stack protector, stack canary, Retguard (OpenBSD)
- Retguard:
  - Při vstupu do funkce zakóduje návratovou adresu
  - ESP se dá považovat za náhodné – je těžké ho uhádnout
  - Před návratem se návratová adresa obnoví XORem
  - Pokud útočník přepsal návratovou adresu, obnovou se jeho adresa znehodnotí → systém „spadne“

Retguard – příklad:

`main:`

```
push    $0x3
push    $0x1
call    500 <func>
pop     %eax
pop     %edx
ret
```

`func:`

```
xor    (%esp),%esp ; zakoduj
sub    $0x10,%esp
mov    0x18(%esp),%eax
add    0x14(%esp),%eax
mov    %eax,0xc(%esp)
mov    0xc(%esp),%eax
add    $0x10,%esp
xor    (%esp),%esp ; obnov
ret
```

# Závěr

- Bezpečnost je důležitým aspektem každého počítačového systému
- V budoucnosti bude její důležitost narůstat
- Systémy (nejen operační) jsou tak bezpečné, jak bezpečný je nejslabší článek
  - I ta nejméně důležitá knihovna používaná vaším programem může obsahovat kritickou zranitelnost
  - I operační systém obsahuje mnoho komponent, které nepoužíváte, ale útočníkům pomohou
- Útočníci jsou velmi kreativní a vynalézaví lidé
- Pokud se jim chcete bránit, musíte umět myslet jako oni

## Reference

- Využili jsme některé materiály licencované pod CC BY 3.0 „Courtesy of Gernot Heiser, UNSW Sydney“.

# B4B35OSY: Operační systémy

## Lekce 9: Vstup/výstup, ovladače

Michal Sojka

[michal.sojka@cvut.cz](mailto:michal.sojka@cvut.cz)



March 9, 2021

# Osnova

## 1 Úvod

## 2 Úložiště

- Jak funguje hardware úložiště?
- Přístup k datům, stránková cache
- Disková pole

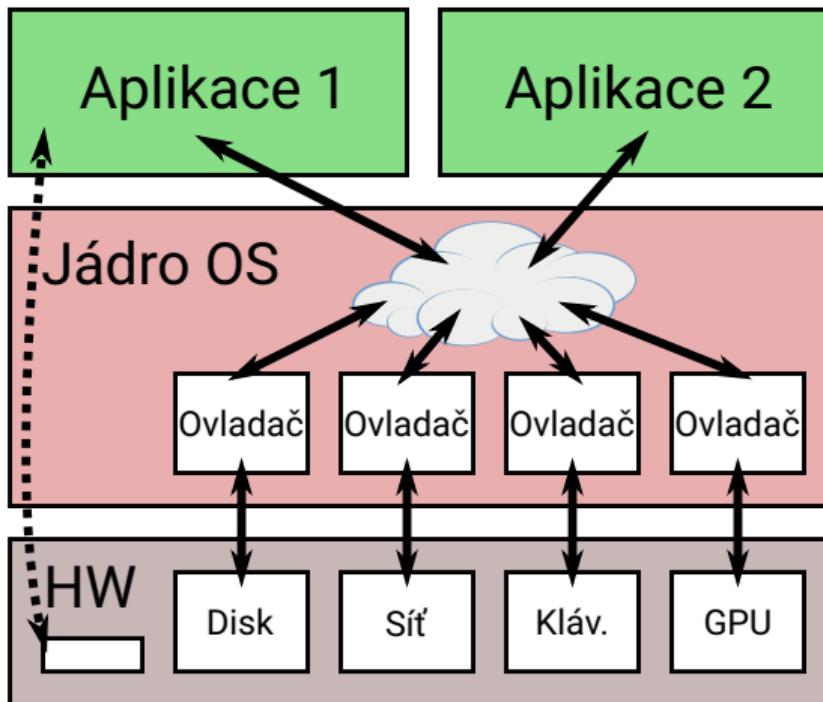
## 3 Síťová rozhraní

- Hardware
- Příjem a odesílání dat
- Rozvrhování rámců

## 4 Ovladače

- Linux
- Windows
- Ovladače v uživatelském prostoru

# Vstup a výstup v OS



- Takto to vypadá v OS s monolitickým jádrem (Linux, Windows)
  - Pro přístup k HW musí aplikace využívat služby jádra
  - Privilegované aplikace (např. root v Linuxu) mohou přistupovat k některému HW přímo (viz sekci Ovladače v uživatelském prostoru)
- V OS s μ-jádry jsou ovladače většinou samostatné procesy mimo jádro, tedy také v uživatelském prostoru

# Vstup a výstup

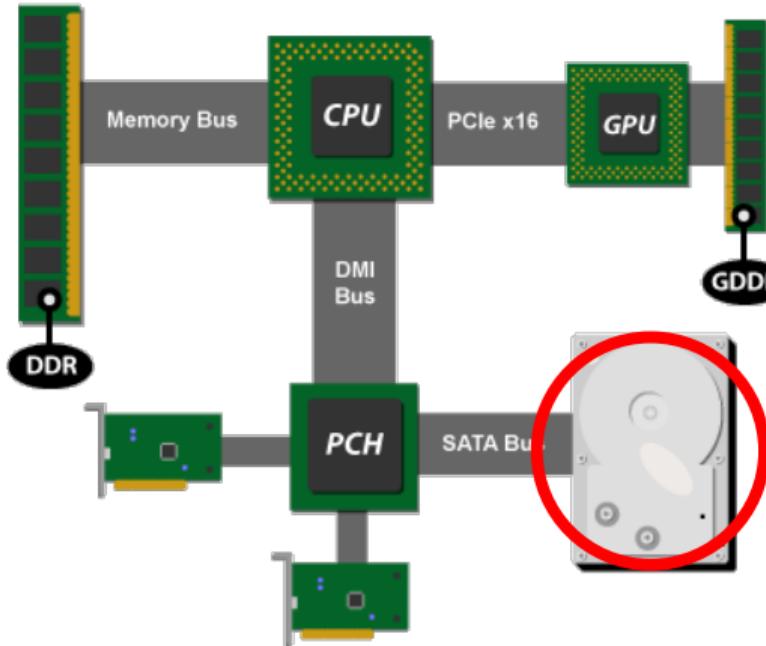
## Input/Output (I/O)

- Způsob, jak počítač komunikuje s okolním světem
  - Datová úložiště (disky)
  - Sítě
  - Klávesnice, monitor, ...
- Uživatelská aplikace nemá přímý přístup k periferiím (HW)
  - Aplikace, která nepoužívá služby jádra OS může pouze číst a zapisovat do (virtuální) paměti
- Pro přístup k periferiím musí používat služby OS, které
  - zajišťují „bezpečné“ **sdílení** periferií mezi aplikacemi a
  - **abstrahují** hardwarové detaily a poskytují jednotné API pro všechny periferie stejné třídy.
  - K tomu využívají služeb **ovladačů zařízení**, které naopak řeší všechny detaily práce s konkrétním hardwarem.

# Úložiště

- HW pro ukládání velkého množství dat
- Není možné číst data po jednotlivých bytech, ale po tzv. blocích či sektorech
- Pevný disk – velikost bloku 512 B, 4 kB, ...
  - Rotační
  - Solid-state (SSD)
- Flash paměť – někdy lze číst po bytech, ale mazat jde jen po blocích – typicky 128 kB
  - Typicky v embedded zařízeních
  - Základem pro SSD disky

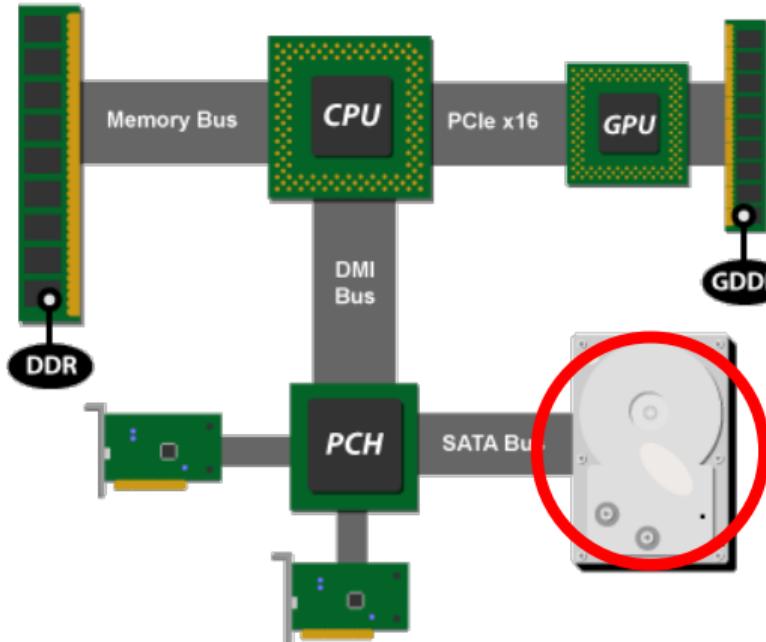
# Model HW



- Pevný disk je malý počítač, který komunikuje s hlavním CPU pomocí sběrnice.
- Přístup k disku je řádové pomalejší než přístup k paměti
- CPU posílá příkazy, disk je autonomně vykonává
- Disk využívá tzv. *Direct Memory Access* (DMA), také označovaný jako *Bus Master*.
  - Data proudí do paměti bez zásahu software v CPU

Platforma Intel's P55. Zdroj: ArsTechnica

# Model HW



- Typické příkazy:
  - Ulož do sektorů 123456–123460 data z paměti na adresu 0x2f003200
  - Načti 32 sektorů počínaje č. 7654 a ulož je do paměti na adresu 0x302f1200
- Disky umí zpracovávat víc příkazů najednou (typicky 32)
  - Interně provádí optimalizace (např. změna pořadí vykonávání či slučování požadavků).
  - O dokončení operace je CPU (přesněji ovladač běžící na CPU) informován přerušením.

Platforma Intel's P55. Zdroj: ArsTechnica

## Přístup aplikací k úložišti

- Aplikace typický nepřistupují k úložišti přímo, ale skrze **souborový systém** (viz příští přednášku)
- OS optimalizuje přístup k úložišti:
  - Spravuje vyrovnávací paměť pro rychlejší přístup k datům na disku
  - OS sám předem načítá data o kterých předpokládá, že budou brzy potřeba
  - Pro pomalé rotační disky:
    - Slučuje požadavky aplikací do větších
    - Rozvrhuje, kdy který požadavek vykonat – optimalizace přejezdů hlaviček – tzv. IO scheduler.

# Stránková vyrovnávací paměť

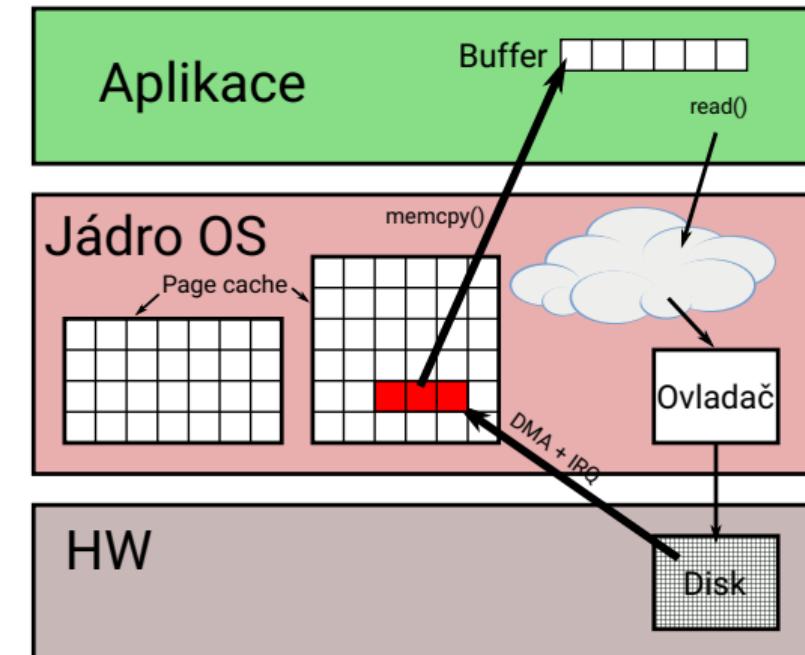
Page cache – název používaný Linuxem pro vyrovnávací paměť disku

- Data čtená z disku resp. zapisovaná na disk jsou uchovávána v paměti pro případné další použití
- OS se snaží využít veškerou volnou paměť jako diskovou cache
- Spravována po stránkách (4 kB)
  - I když starší disky používaly 512 B sektory, OS (téměř) vždy načítá celé 4 kB.

# Čtení a zápis

## ■ Čtení z disku:

- 1 Aplikace zavolá `read()`
- 2 Jádro OS (JOS) přepošle požadavek správnému ovladači
- 3 Ovladač disku pošle příkaz pro načtení dat a uložení do page cache
- 4 Disk sám o sobě posílá data do paměti (DMA) a o dokončení informuje přerušením (interrupt request, IRQ)
- 5 V reakci na IRQ, JOS zkopíruje data z page cache do paměti aplikace



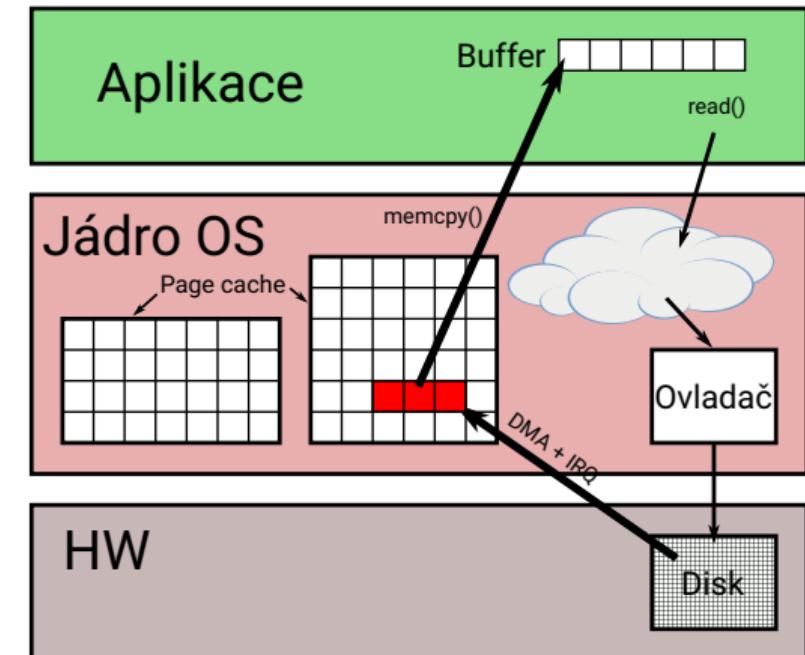
# Čtení a zápis

## ■ Zápis na disk:

- 1 Aplikace zavolá write()
- 2 JOS data zkopíruje z aplikace do page cache a write() se vrátí.
- 3 Čas od času JOS zapisuje „špinavé stránky“ na disk. V Linuxu označováno jako „writeback“.
  - Zápis se dá vynutit systémovým voláním fsync() (Linux)

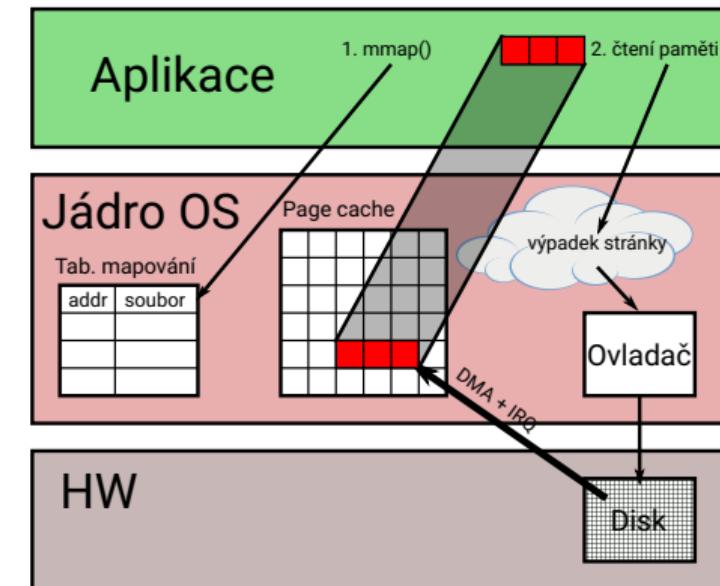
Pozn.: fsync() vs. fflush()

**fsync()** ukládá data z page cache na disk,  
**fflush()** posílá data z bufferu aplikace (schovaný v libc) do jádra (page cache v případě práce se soubory soubory).



# Čtení a zápis bez zbytečného kopírování dat

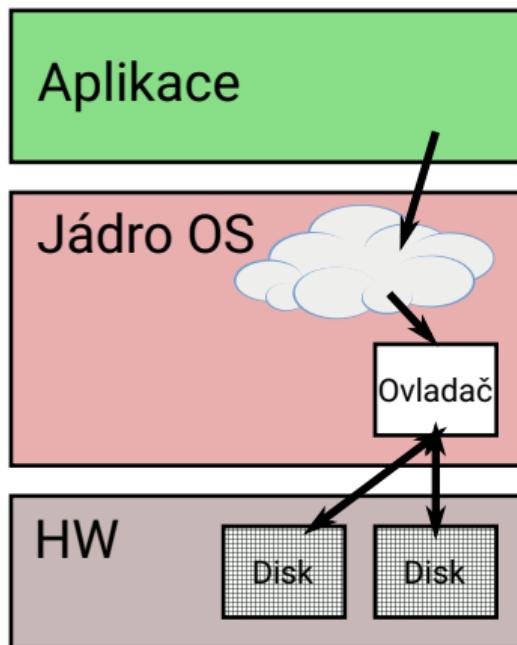
- Aplikace může požádat JOS, aby „namapoval“ stránky diskové cache do jejího adresního prostoru (v UNIXu systémové volání mmap())
  - Při prvním přístupu k paměti vrácené funkcí mmap dojde k výjimce (výpadku stránky), protože nic ještě není namapováno
    - 1 JOS se podívá do tabulky mapování (pro Vás viditelná v /proc/<PID>/maps), aby zjistil, jaký soubor je potřeba načíst a načte data z disku do stránkové cache
    - 2 Poté modifikuje stránkovací tabulku procesu a vrátí se z obsluhy výjimky na instrukci, která výjimku způsobila
    - 3 Tentokrát se instrukce provede úspěšně a aplikace pokračuje
- Zápis se provádí stejně jako čtení prostým zápisem do namapované paměti
- Čas od času JOS zapisuje „špinavé stránky“ na disk.
- Může aplikace zjistit (nebo zajistit), ?



- Pouze při použití `msync()` máme jistotu, že jsou data uložena na disku (pro případ výpadku napájení)
- Sdílení dat jednoho souboru mezi procesy se uskutečňuje prostřednictvím page-cache a není vázáno na uložení na disk

# Disková pole

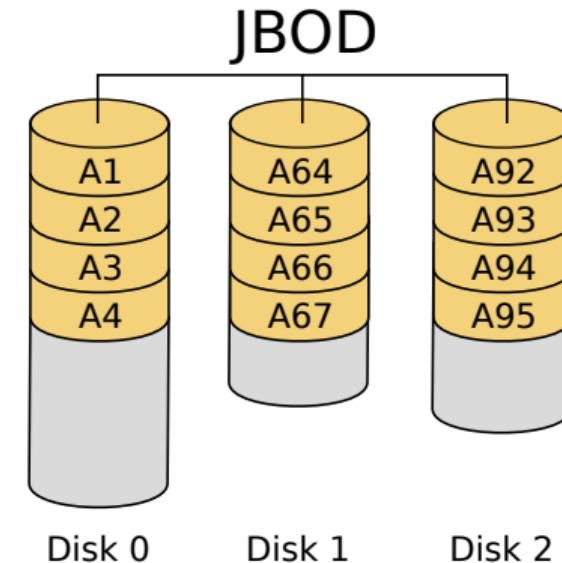
RAID – Redundant Array of Independent Disks



- Pokud se disk porouchá, přijdeme o (cenná) data
- Redundance – data jsou uložená na více místech najednou
- Možnost implementace v HW nebo v SW (OS)
- Rychlosť SW implementace – čtení typicky rychlejší (paralelní čtení z více disků), zápis o něco pomalejší.
- Nahradí RAID zálohování dat?
  - Požár v serverovně – záloha na jiném místě
  - Administrátor omylem smaže data

# Typy diskových polí

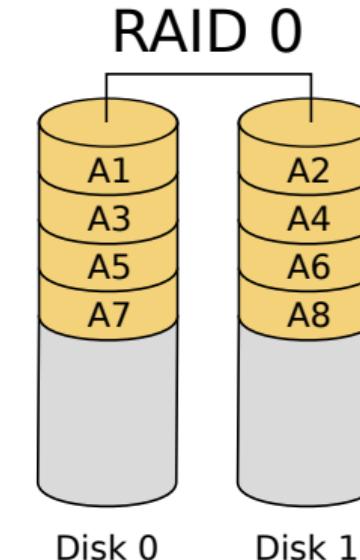
- RAID0 – spojení více disků do jednoho virtuálního (bez redundancy)
- RAID1 – zrcadlení, efektivita: 50%
- RAID5 – prokládání dat a parita, min. 3 disky, toleruje ztrátu jednoho disku, efektivita  $\frac{n-1}{n}$   
 $A_p = A_1 \oplus A_2 \oplus A_3 \text{ (xor)}$   
Při poruše 1. disku:  
 $A_1 = A_p \oplus A_2 \oplus A_3$
- RAID6 – toleruje ztrátu dvou disků



Autor: en:User:Cburnett – Vlastní dílo, CC BY-SA 3.0

# Typy diskových polí

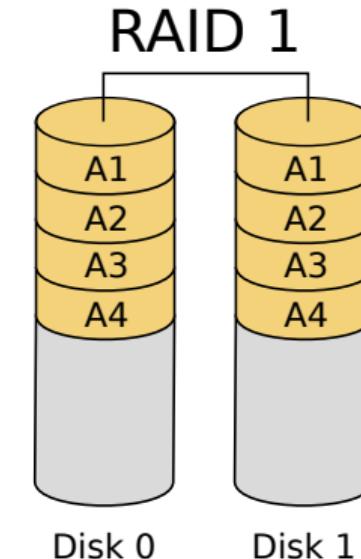
- RAID0 – spojení více disků do jednoho virtuálního (bez redundancy)
- RAID1 – zrcadlení, efektivita: 50%
- RAID5 – prokládání dat a parita, min. 3 disky, toleruje ztrátu jednoho disku, efektivita  $\frac{n-1}{n}$   
 $A_p = A_1 \oplus A_2 \oplus A_3 \text{ (xor)}$   
Při poruše 1. disku:  
 $A_1 = A_p \oplus A_2 \oplus A_3$
- RAID6 – toleruje ztrátu dvou disků



Autor: en:User:Cburnett – Vlastní dílo, CC BY-SA 3.0

# Typy diskových polí

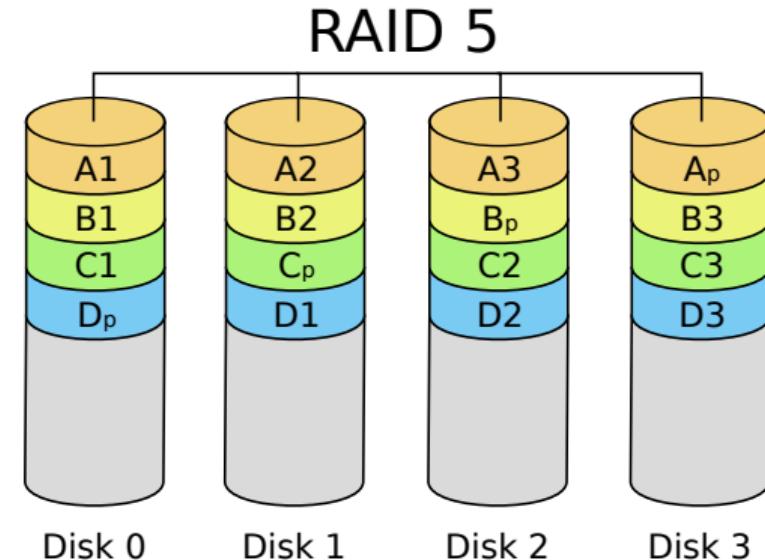
- RAID0 – spojení více disků do jednoho virtuálního (bez redundancy)
- RAID1 – zrcadlení, efektivita: 50%
- RAID5 – prokládání dat a parita, min. 3 disky, toleruje ztrátu jednoho disku, efektivita  $\frac{n-1}{n}$   
 $A_p = A_1 \oplus A_2 \oplus A_3$  (xor)  
Při poruše 1. disku:  
 $A_1 = A_p \oplus A_2 \oplus A_3$
- RAID6 – toleruje ztrátu dvou disků



Autor: en:User:Cburnett – Vlastní dílo, CC BY-SA 3.0

# Typy diskových polí

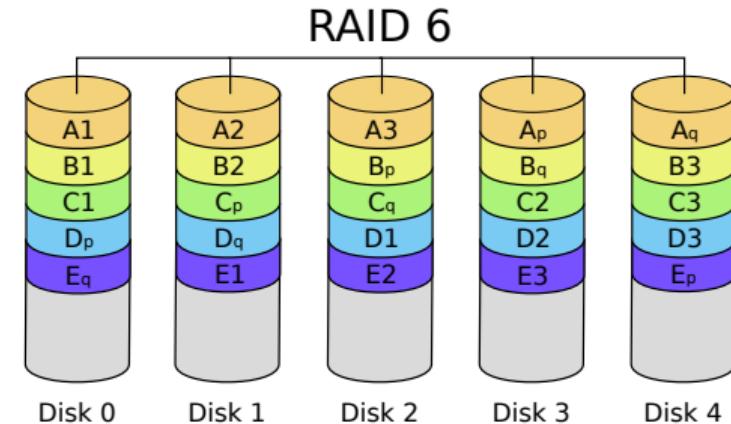
- RAID0 – spojení více disků do jednoho virtuálního (bez redundancy)
- RAID1 – zrcadlení, efektivita: 50%
- RAID5 – prokládání dat a parita, min. 3 disky, toleruje ztrátu jednoho disku, efektivita  $\frac{n-1}{n}$   
 $A_p = A_1 \oplus A_2 \oplus A_3$  (xor)  
 Při poruše 1. disku:  
 $A_1 = A_p \oplus A_2 \oplus A_3$
- RAID6 – toleruje ztrátu dvou disků



Autor: en:User:Cburnett – Vlastní dílo, CC BY-SA 3.0

# Typy diskových polí

- RAID0 – spojení více disků do jednoho virtuálního (bez redundancy)
- RAID1 – zrcadlení, efektivita: 50%
- RAID5 – prokládání dat a parita, min. 3 disky, toleruje ztrátu jednoho disku, efektivita  $\frac{n-1}{n}$   
 $A_p = A_1 \oplus A_2 \oplus A_3$  (xor)  
 Při poruše 1. disku:  
 $A_1 = A_p \oplus A_2 \oplus A_3$
- RAID6 – toleruje ztrátu dvou disků

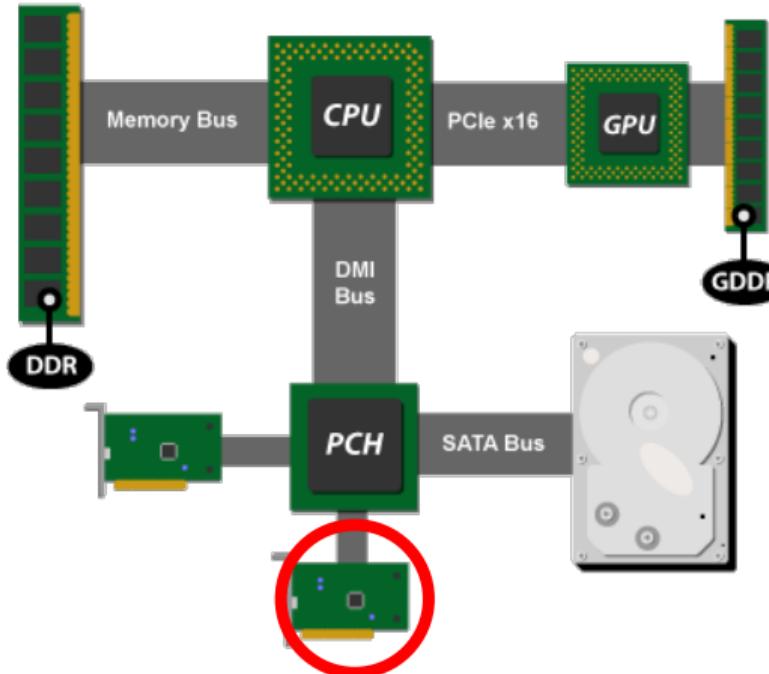


Autor: en:User:Cburnett – Vlastní dílo, CC BY-SA 3.0

# Sítě

- Síťový subsystém OS řeší mnoho různých aspektů:
  - Ethernet, Wi-Fi, Bluetooth, CAN, ...
  - Routování
  - Virtuální síť – VPN, ...
- Ethernet představuje základní model sítě používaný OS
  - Základní funkce všech technologií jsou stejné: **posílání a příjem rámců**
  - Jednotlivé síťové technologie se liší především nastavováním parametrů (WiFi: SSID, Ethernet: bitrate,
- OS reprezentuje síťový HW pomocí tzv. síťových rozhraní
- Sítě jsou velmi rychlé – dnes až 100 Gbps
- Síťový subsystém OS musí být velmi efektivní, aby OS nebyl úzkým hrdlem
- Úložiště a sítě mají z pohledu OS mnoho společného
  - Do nedávna nebyla efektivita diskového substitutu důležitá, ale s nástupem rychlých SSD disků nabývá na důležitosti a síťování je zde inspirací

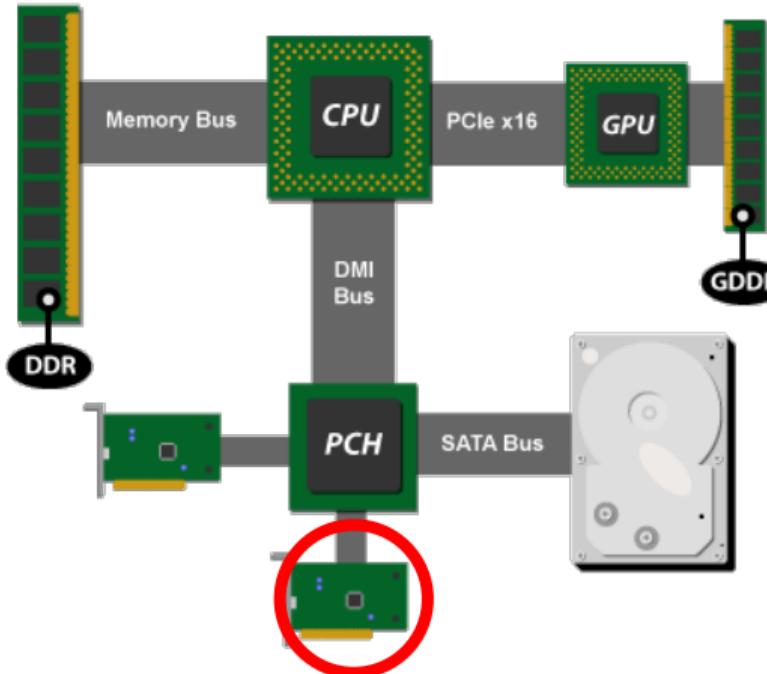
# Sítový hardware



- Sítové rozhraní je malý počítač, který komunikuje s hlavním CPU pomocí sběrnice.
- CPU posílá příkazy, síťové rozhraní je autonomně vykonává
- Používá se tzv. *Direct Memory Access (DMA)*, také označovaný jako *Bus Master*.
  - Data proudí z/do paměti bez zásahu software v CPU

Platforma Intel's P55. Zdroj: ArsTechnica

# Sítový hardware

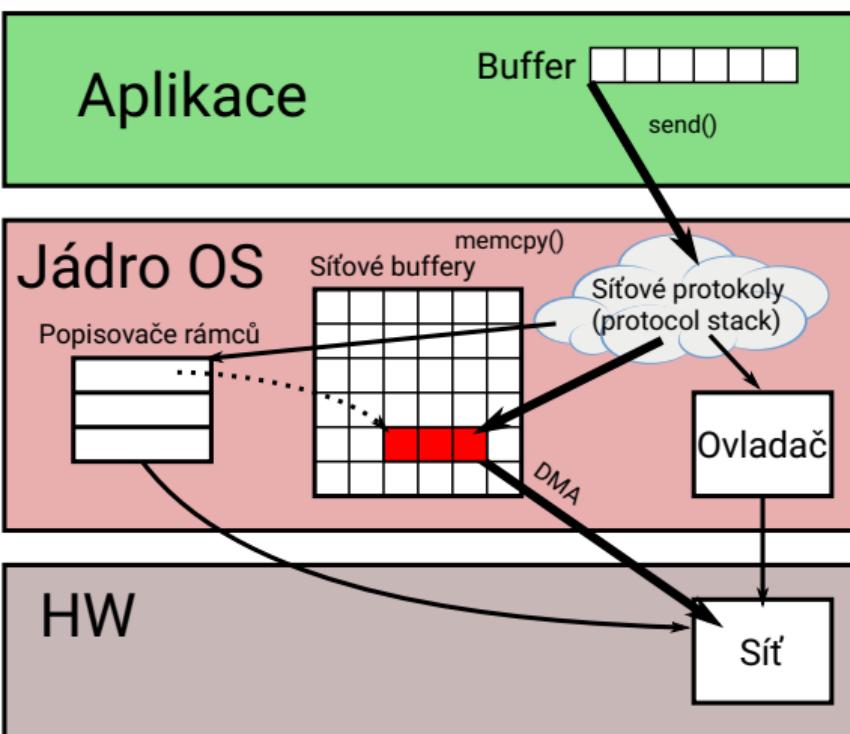


## ■ Typické „příkazy“:

- Pošli rámec, který je uložený na adresě 0x2f003200.
- Pokud přijmeš rámec, ulož ho na adresu 0x302f1200.
- Implementováno pomocí tabulky popisovačů rámců (packet descriptor table)
  - ovladač vytvoří v paměti tabulkou ukazatelů na rámce a síťové rozhraní se do ní „kouká“ při příjmu či odesílání rámce.

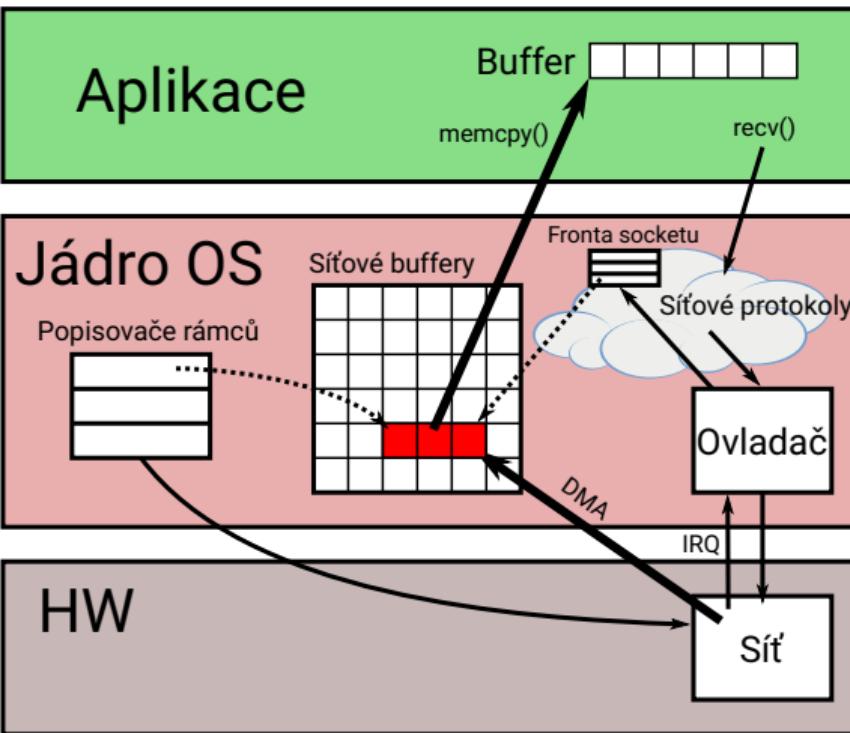
Platforma Intel's P55. Zdroj: ArsTechnica

# Odesílání dat aplikacemi



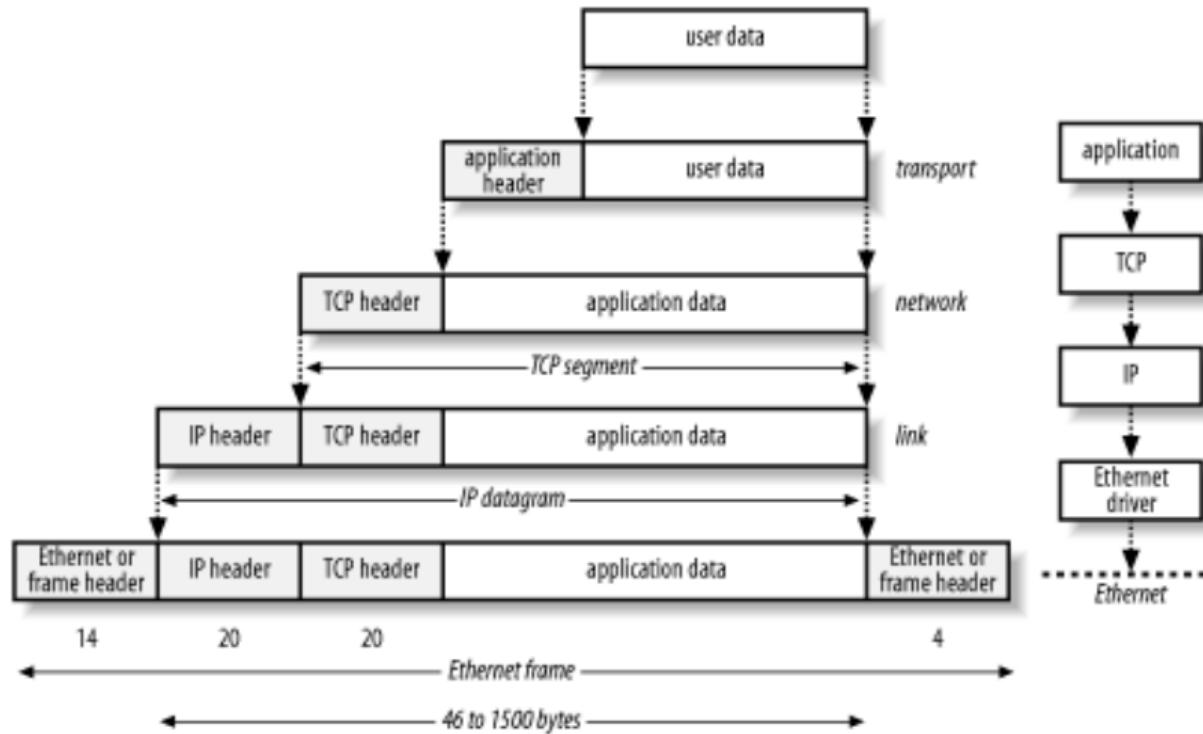
- 1 Aplikace zavolá send()/write()
- 2 JOS si zkopiřuje odesílaná data do síťových bufferů
- 3 JOS (tzv. protocol stack) přidá k aplikačním datům potřebné hlavičky a upozorní ovladač
- 4 Ovladač upraví tabulku popisovačů rámců, a dá vědět (jak?zápisem do registru v síťovém HW) síťovému HW, že se tabulka popisovačů změnila.
- 5 Síťový HW začne číst data z paměti (DMA) a odešle je.

# Příjem dat aplikacemi



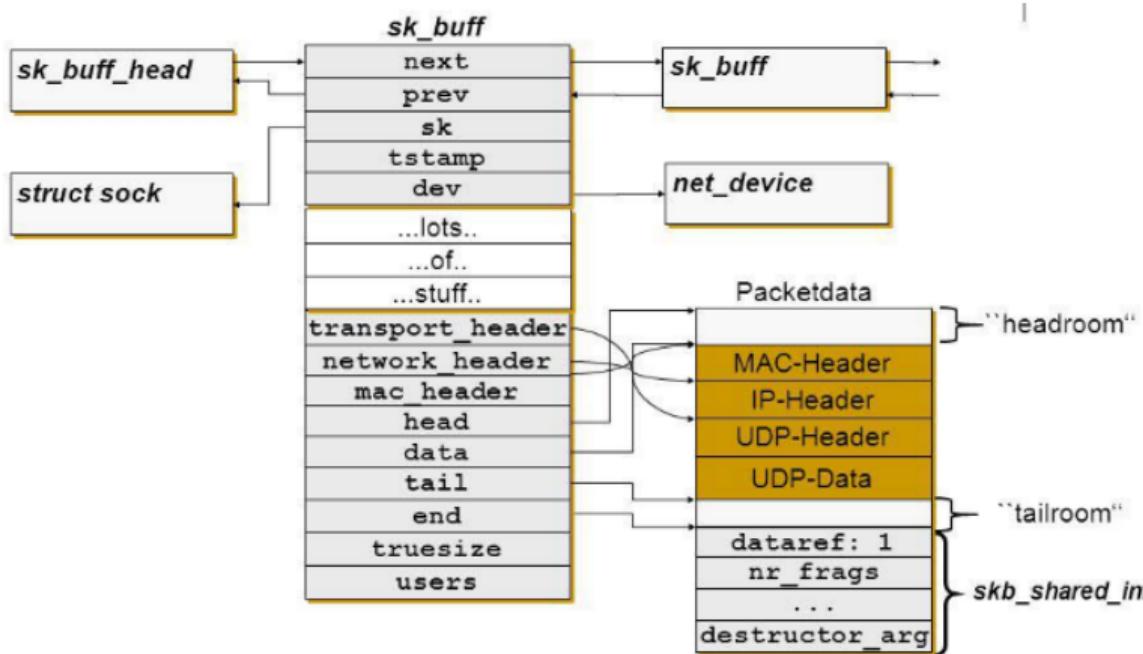
- 1 Aplikace zavolá recv()/read()
- 2 JOS zkонтroluje zda fronta socketu obsahuje nějaká přijatá data. Pokud ano, pokračuje se krokem 6, v opačném případě se volající vlákno zablokuje a čeká.
- 3 Síťové rozhraní autonomně ukládá přijímané rámce do paměti (DMA).
- 4 Po dokončení příjmu je upozorněn ovladač (přerušení) a ten pak aktivuje zpracování rámce síťovými protokoly.
- 5 Poté JOS zařadí rámec do fronty patřičného soketu.
- 6 JOS přijatá data nakopíruje ze síťových bufferů v jádře do aplikačního bufferu a systémové volání se vrátí do aplikace.

# Sítové protokoly



# Datová struktura pro práci se sítovými rámci

struct skbuff v Linuxu



- Možnost přidávat hlavičky před data, bez nutnosti jejich kopírování
- Scatter-gatter DMA – hardware si umí sestavit rámeček „za běhu“ z více částí

# Příjem a odesílání dat bez kopírování

## Zero-copy networking

- Podobný „trik“, jako s diskovou vyrovnávací pamětí
- `socket(AF_PACKET, ...)` + `mmap()`
- Síťový HW přijímá/odesílá rámce rovnou do/z paměti kontrolované aplikací
- Nevýhody:
  - Aplikace si musí sama řešit přidávání a odebrání hlaviček
  - Aplikace nesmí modifikovat rámce (např. kvůli chybě v programu) , které jsou v procesu odesílání.

# Rozvrhování rámců při odesílání

## Traffic scheduling/control

- Další z činností, kterou řeší síťový stack OS
- Prioritizace interaktivní komunikace
- Spravedlivé rozdělení šířky pásma mezi uživatele (zákazníky)
- Problém zvaný „buffer-bloat“
  - Ovladač může do odesílací fronty (popisovač rámců k odeslání) uložit velké množství rámců.
  - Síťový HW odesílá rámce v pořadí, v jakém jsou tam uvedeny.
  - Pokud je na konci fronty rámec, který by měl být odeslán přednostně, musí dlouho čekat.
  - Řešení:
    - Fronta ovladače se udržuje krátká, aby kritické rámce mohly „předbíhat“
- Moderní síťový hardware implementuje více front pro odesílání (i příjem)
  - Rámce jsou rozvrhovány (i) v hardwaru – výběr fronty
  - Využívá se ve vícejádrových systémech, kde má každé jádro samostatnou frontu a není potřeba v ovladači ztrácat čas synchronizací (mutex) mezi různými CPU
  - Někdy lze využít i k prioritizaci rámců – každá fronta má jinou prioritu

# Ovladač zařízení

Device driver

- Software, který
  - 1 ovládá konkrétní zařízení (disk, síťová karta, GPU, ...) a
  - 2 zbytku OS nabízí jednotné rozhraní (API)
- Se zařízením typicky komunikuje pomocí do paměti mapovaných registrů
  - V Linuxu viz příkaz `lspci -v`

```
01:00.0 Network controller: Intel Corporation Wireless 8260 (rev 3a)
    Subsystem: Intel Corporation Wireless 8260
    Flags: bus master, fast devsel, latency 0, IRQ 129
    Memory at ef100000 (64-bit, non-prefetchable) [size=8K]
    Capabilities: <access denied>
    Kernel driver in use: iwlwifi
    Kernel modules: iwlwifi
```
- Obsluhuje přerušení od zařízení (IRQ)

# Spolehlivost ovladačů

- Ovladače bývají nejméně spolehlivou částí jádra OS
  - Chyba kdekoli v jádře OS (tedy i v ovladači) může způsobit nestabilitu celého systému
  - Ne každý programátor ovladačů rozumí všem potřebným detailům
  - Ovladače se nedají testovat, pokud není k dispozici konkrétní HW
  - Velmi špatně se testuje obsluha chybových stavů, protože je potřeba donutit HW, aby signalizoval chybu
  - Microsoft zavedl povinné digitální podepisování ovladačů, aby měl částečnou kontrolu nad jejich kvalitou
- Dnešní OS umožňují, aby některé ovladače běžely v uživatelském prostoru (jako aplikace), podobně jako je to běžné u μ-jader (viz např. UIO dále)

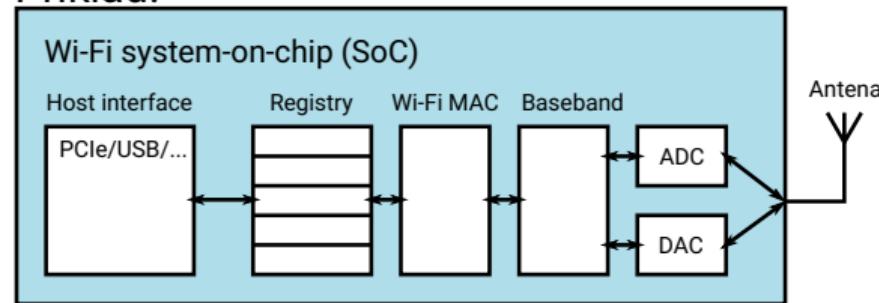
# Příklad – ovladač klávesnice

- 1 Aplikace zavolá **getch()**/**scanf()**/... na standardní vstup
- 2 libc vyvolá systémové volání **read()** na deskriptoru souboru 0 (stdin)
- 3 Standardní vstup je připojen k terminálu (klávesnice + obrazovka) – JOS tedy předá požadavek na vstup **ovladači klávesnice**
  - Ovladač klávesnice spravuje frontu stisknutých znaků (kláves)
- 4 Pokud je fronta prázdná, ovladač **uspí volající vlákno**, jinak se pokračuje krokem 7
  - Interně k tomu použije semafor – vlákno přidá do fronty semaforu
  - Poté zavolá plánovač, aby vybral jiné vlákno, které poběží
- 5 Po stisku klávesy HW vyvolá **přerušení**
- 6 Ovladač klávesnice přerušení obslouží:
  - Přečte z HW (registru) jaká byla stisknuta klávesa a uloží ji do fronty
  - Zavolá operaci up/post na semafor
- 7 Uspané vlákno aplikace se **probudí** (je stále v jádře), vyčte z fronty ovladače stisknuté znaky a zkopíruje je do bufferu v aplikaci.
- 8 Provede se **návrat** ze systémového volání zpět do aplikace, funkce **getch/scanf** se dokončí.

# Variabilita a složitost HW

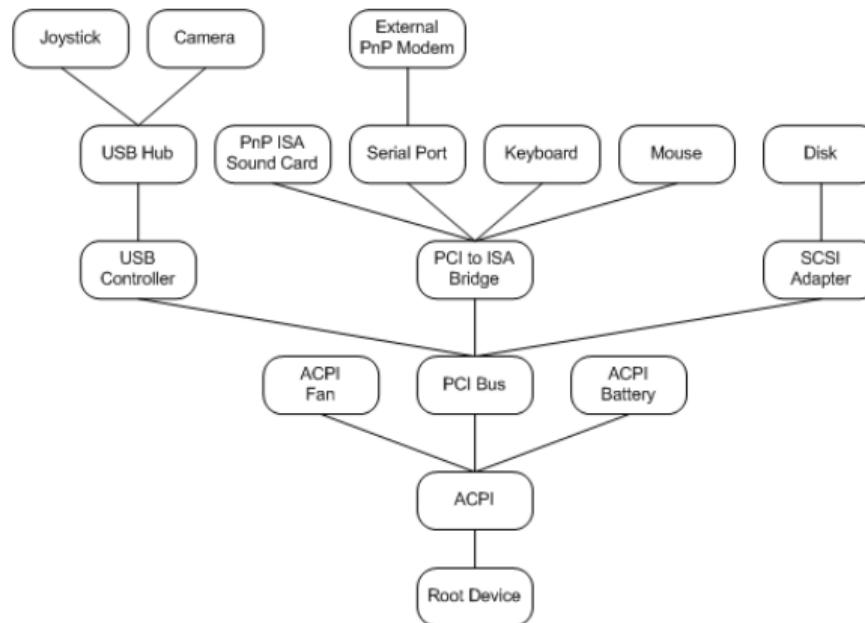
- Dnešní hardware je složitý, zařízení mohou obsahovat stovky či tisíce registrů
- I vývojáři HW mají v oblibě „Ctrl-C, Ctrl-V“ – jeden čip existuje v mnoha variantách, ale všechny mají téměř stejně registry
  - Např. Wi-Fi čip – jedna verze se připojuje k PCIe, jiná k USB
- Struktura ovladačů je modulární – chceme ovladač napsat jednou a používat pro všechny varianty čipu

Příklad:



# Hierarchie ovladačů

## Topologie hardwaru



- Ovladače reflektují topologii HW
- Každý uzel má vlastní ovladač nezávislý na okolí
- Plug-and-Play (PnP)
  - Ovladač sběrnice (USB, PCI) detekuje připojená zařízení a automaticky načte potřebný ovladač zařízení

# Ovladače v Linuxu

- Aplikace mohou s ovladači komunikovat různými způsoby:
  - Nepřímo – např. přes síťové API, práci se soubory, stdin/out
  - Přímo – většina zařízení je reprezentována jako speciální soubor v adresáři /dev (např. sériová linka /dev/ttyUSB0).
  - Pomocí knihoven – aplikace často k souborům v /dev přistupují pomocí knihoven (např. libusb), které nabízejí vyšší úroveň abstrakce, než API OS.
- Ovladač poskytuje aplikacím následující operace (nízkoúrovňové API) pro přímý přístup k ovlaačům:
  - **open** – slouží pro „navázání spojení“ aplikace s ovladačem
  - **read** – čtení dat ze zařízení (např. zkuste si spustit *hexdump /dev/input/mice*)
  - **write** – zápis dat do zařízení (např. *tty; echo XXX > /dev/pts/3*),
  - **ioctl** – vše ostatní, co není čtení či zápis, často nastavování (*man ioctl\_list, ioctl\_tty, ...*)
  - **close** – ukončení komunikace s ovladačem

# Nejjednodušší ovladač

## Ovladač virtuálního zařízení /dev/null

```
#define NULL_MAJOR 1          /* dev major number */
#define NULL_MINOR 3           /* dev minor number */

ssize_t read_null(struct file *file, char *buf, size_t count, loff_t *ppos)
{
    return 0;
}

ssize_t write_null(struct file *file, const char *buf, size_t count, loff_t *ppos)
{
    return count;
}

const struct file_operations null_fops = {
    .read  = read_null,
    .write = write_null,
};

void init()
{
    register_chrdev(NULL_MAJOR, NULL_MINOR, "null", &null_fops)
}
```

Trochu zjednodušeno; skutečná implementace ovladače /dev/null: <https://elixir.bootlin.com/linux/v5.9.10/source/drivers/char/mem.c#L673>

# Druhý nejjednodušší ovladač

## Ovladač virtuálního zařízení /dev/zero

```
#define ZERO_MAJOR 1          /* dev major number */
#define ZERO_MINOR 5           /* dev minor number */

ssize_t read_zero(struct file *file, char *buf, size_t count, loff_t *ppos)
{
    memset(buf, 0, count);
    return count;
}

ssize_t write_zero(struct file *file, const char *buf, size_t count, loff_t *ppos)
{
    return count;
}

const struct file_operations zero_fops = {
    .read  = read_zero,
    .write = write_zero,
};

void init()
{
    register_chrdev(ZERO_MAJOR, ZERO_MINOR, "zero", &zero_fops)
}
```

# Komunikace mezi ovladači (Linux)

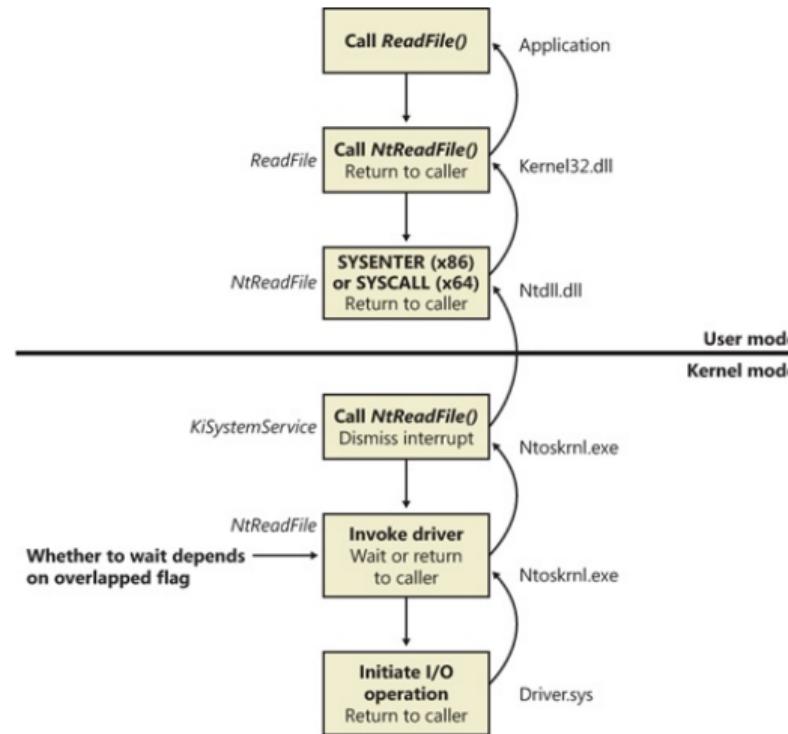
- Linux je monolitické jádro.
- Jednotlivé ovladače se volají vzájemně úplně stejně, jako se volají funkce v uživatelských aplikacích.
- Často se funkce nevolá přímo, ale přes ukazatel
  - Např. Každý ovladač si registruje ukazatel na funkci, která se má vyvolat, když aplikace zavolá `read()` (viz `struct file_operations` na předchozích slidech).
- Data se předávají skrze argumenty funkcí (buď přímo nebo pomocí ukazatelů).

# Přístup k ovladačům ve Windows

- Z pohledu aplikace konceptuálně podobné Linuxu:

	<b>Linux</b>	<b>Windows</b>
<b>Otevření ovladače</b>	open	CreateFile
<b>Operace s ovladačem</b>	read, write, ioctl	ReadFile, DeviceIoControl, ...
<b>Uzavření ovladače</b>	close	CloseHandle
<b>Jmenný prostor</b>	/dev/	\.\.
<b>Příklad</b>	/dev/ttyUSB0	\.\.COM6

# Přístup k ovladačům ve Windows

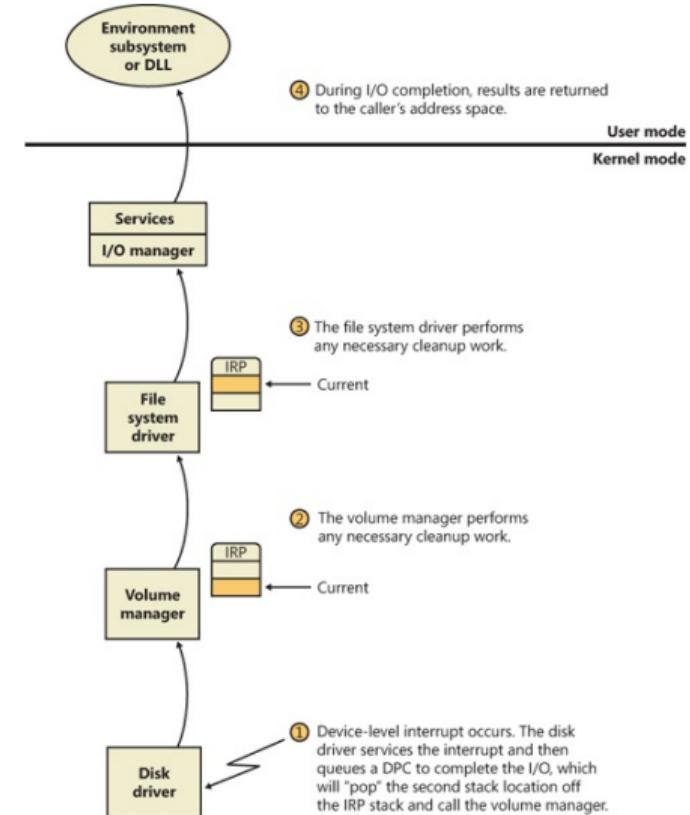
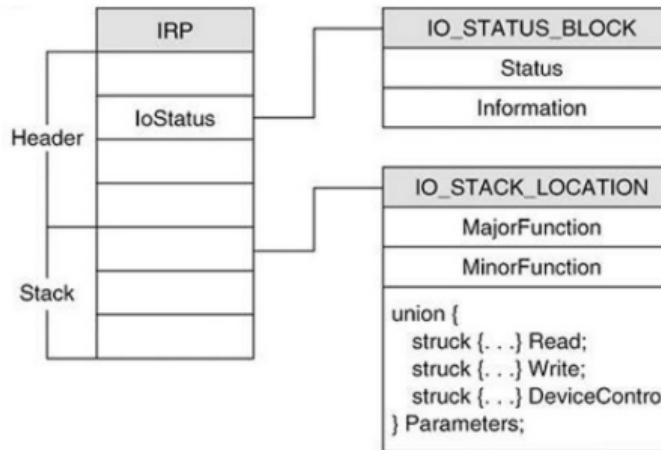


Source: Microsoft

# Komunikace mezi ovladači v jádře Windows

- Ovladače ve Windows nepoužívají přímé volání funkcí, ale komunikují pomocí předávání zpráv
- Windows Driver Model je navržen tak, aby bylo teoreticky možné pouštět ovladače v oddělených adresních prostorech
- Kvůli rychlosti ale běží většina ovladačů v jednom monolitickém adresním prostoru jádra.
- Zprávy, které si ovladače vyměňují, se nazývají **I/O request packet (IRP)**

# Cesta IRP jádrem



- IRP se alokuje jen jednou
- Každý ovladač „po cestě“ má svůj slot
- File system vyplní slot pro volume manager a pošle IRP dál.
- Po dokončení požadavku IRP „cestuje“ zpět (obr. vpravo).

# Ovladače v uživatelském prostoru

- Chyba v ovladači může způsobit pád systému
- Nekvalitní ovladače jsou také zdrojem mnoha bezpečnostních problémů
- Ovladače v uživatelském prostoru:
  - Podporovány jak Linuxem (UIO), tak Windows
  - Spouštěny jako běžná aplikace
  - Přístup k registrům HW: Pomocí vlání *mmap()*
  - Obsluha přerušení – OS upozorní aplikaci pokud nastalo přerušení
    - UIO subsystém v Linuxu:

```
int uio = open("/dev/uio0", ...);
read(uio, ...); // waits for interrupt
handle_interrupt();
```
    - Při chybě ovladače ho lze jednoduše restartovat
    - Ostatní aplikace nevolají ovladač pomocí systémových volání, ale pomocí meziprocesní komunikace (např. fronty zpráv)
- OS založené na mikrojádře mají (téměř) všechny ovladače v uživatelském prostoru

# B4B350SY: Operační systémy

## Souborové systémy

Michal Sojka<sup>1</sup>



2020-11-26

---

<sup>1</sup>[michal.sojka@cvut.cz](mailto:michal.sojka@cvut.cz)

# Obsah I

## 1 Úvod

## 2 Souborové systémy

- Základy
- FAT
- Souborový systém založený na inode

## 3 Žurnálování

## 4 Souborové systémy pro Flash paměti

# Co je souborový systém?

- Způsob organizace dat na pevném disku
- Data uložená v pojmenovaných souborech
- Soubory v adresářích (složkách)
- Hierarchická struktura adresářů

## Terminologie

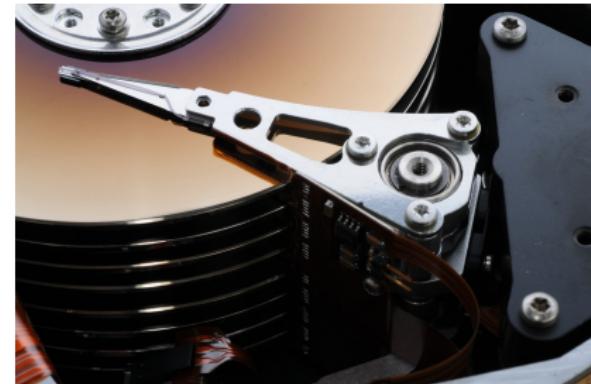
- Data = obsah souborů
- Metadata = pomocné informace ukládané souborovým systémem (bitmapy, inody, superblok, ...)

## Požadavky na souborový systém

- Efektivita (nízká režie) – metadata zaberou méně než X % kapacity disku
- Rychlosť
- Nízká fragmentace – souvisí s rychlosťí (viz dále)
- Spolehlivosť – data mohou být velmi cenná.

# Pevný disk

- Trvalé uložení dat (i bez napájení)
- Rotační, Flash
- Posloupnost bloků (sektorů) určité velikosti
- Každý blok je identifikován číslem
- Oddíly (partitions)
  - Fyzický disk lze rozdělit na víc logických disků
  - Na začátku disku je tabulka definující typ, (jméno), počáteční a koncový sektor oddílu
    - Master Boot Record (MBR) – pozůstatek MS-DOSu, 1. sektor na disku (512 B), obsahuje místo pro 4 oddíly.
    - GUID Partition Table (GPT) – modernější, více informací, „neomezený“ počet oddílů.
  - Většina souborových systémů využívá jeden logický disk



# Otázky

- Jak ukládat adresáře?
- Jak zjistit, ve kterých blocích jsou data daného souboru?
- Jak alokovat bloky na disku při vytváření/zvětšování souborů?
- Jak se vypořádat s chybami a pády systému?
- Jak optimalizovat souborové systémy pro rotační disky a Flash paměti?

# Adresáře

- Adresář je seznam dvojic («jméno souboru», «umístění»)
- Jméno:
  - V UNIXu všechny znaky kromě / a NUL
  - Ve Windows nesmí obsahovat znaky /\:\*"?<>|
- Umístění: kde jsou uložena data daného souboru – viz dále
- Třídění seznamu (položek v adresáři):
  - Seznam není uložen setříděný; třídění provádí až program zobrazující adresář uživateli podle jím zadaných kritérií
    - Třídění podle názvu, data přístupu, typu souboru
    - Pomalé otevírání souborů ve velkých adresářích
  - Vyhledávací B-strom
    - Rychlejší

# Rozložení dat na disku

- Souborový systém definuje **velikost bloku** (např. 4 KiB)
  - Prostor na disku je vždy alokován v násobcích velikosti bloku
- **Superblok** určuje umístění kořenového adresáře a další informace o souborovém systému
  - Vždy na předem známém místě (např. 1. blok na disku)
  - Často uložen ve více kopiích
- Informace o **volných blocích**
  - OS musí mít přehled, který blok je volný a který obsazený
  - Podobné jako v alokátorech paměti – např. freelist
  - Typicky bitová mapa (1 bit na blok)
  - Kopie v paměti pro urychlení přístupu (cache)
- Bloky ukládající **obsah souborů**
  - Existuje mnoho způsobů, jak je organizovat

# Překlad cesty k souboru

- Co se děje při otevírání souboru „/jedna/dva/tři“?
  - 1 Otevře se kořenový adresář „/“ (vždy se ví, kde se najde – superblok)
  - 2 Najde se v něm záznam „jedna“ a zjistí se jeho *umístění*
  - 3 Otevře se adresář „jedna“ a najde se záznam „dva“ a jeho *umístění*
  - 4 Otevře se adresář „dva“, najde se záznam „tři“ a jeho *umístění*
  - 5 Otevře se soubor „tři“
- Procházení cesty a adresářů po cestě může trvat dlouho
  - Proto je volání `open` odděleno od `read / write`
  - Cesta se prochází jen při `open`
  - Položky adresářů se ukládají do vyrovnávací paměti (dentry cache v Linuxu)

# Základní možnosti uložení obsahu souboru

- Obsah souboru je typicky uložen ve více blocích (do jednoho se nemusí vejít)
- Jak se zjistí, které bloky to jsou? Více možností:
  - Soubor je vždy uložen v souvislém úseku bloků
    - Podobné alokaci paměti
    - Rychlý přístup k datům (lokalita)
    - Neflexibilní, způsobuje fragmentaci a nutnost přemisťovat soubory
  - Spojové seznamy
    - Každý blok obsahuje kromě dat i odkaz na další blok, adresář odkazuje na 1. blok souboru
    - Výhodné pro sekvenční přístup k souborům, nevýhodné pro vše ostatní
    - Nemožnost „mapovat“ data z disku přímo do paměti
    - Jeden špatný sektor na disku (porucha) může způsobit „ztrátu“ zbytku souboru
  - Indexové struktury
    - „Indexový blok“ obsahuje ukazatele (čísla bloků) na mnoho jiných bloků
    - Vhodnější pro náhodný přístup, stále poměrně dobré pro sekvenční přístup
    - Může být potřeba použít více indexových bloků

# Souborový systém FAT

## File Allocation Table

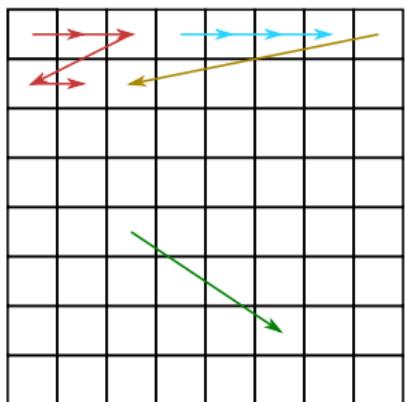
Starý souborový systém s mnoha nedostatky a omezeními.

- Základní jednotka „cluster“ (4–32 KiB)
- FAT12:  $2^{12}$  clusterů, FAT16:  $2^{16}$ , FAT32:  $2^{28}$
- Rozložení disku



- MBR – master boot record (info o soub. systému, tj. velikost, jméno, počet kopií FAT tabulek atd.)
- FAT1, 2 – dvě kopie FAT tabulky (redundance)

# Tabulka FAT



- Jedna položka FAT tabulky má 12/16/32 bitů a odpovídá jednomu clusteru na disku
- Hodnota položky udává číslo následujícího clusteru (konec šipky) nebo -1 značící konec souboru.
- Číslo 1. clusteru se najde v položce adresáře
- Pro urychlení přístupu je tabulka uchovávána v paměti

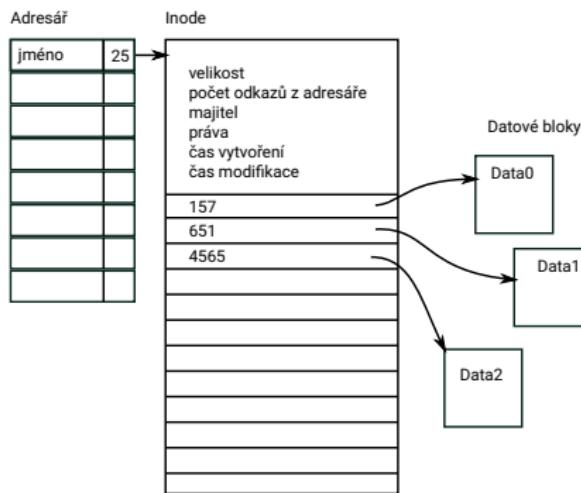
## Nevýhody

- Fragmentace
- Omezená velikost (např.  $2^{32} \times 4\text{KiB}$ )
- Nutnost procházet FAT položky sekvenčně (zpomaluje náhodný přístup u velkých souborů)

## Indexový souborový systém

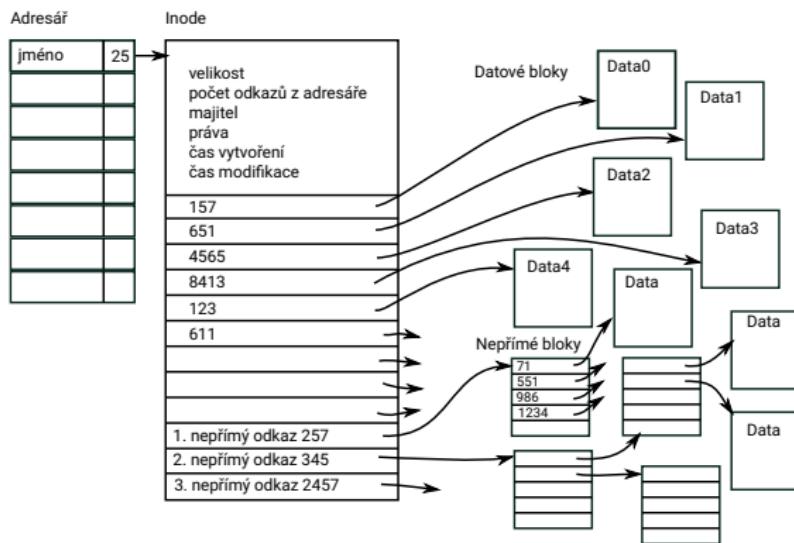
Základ mnoha UNIXových souborových systémů (např. Linuxový ext2 – ext4).

- Metadata o jednotlivých souborech jsou uložena v datové struktuře zvané **inode**.
  - Položka adresáře obsahuje kromě jména souboru i číslo (pořadí) inode
  - inode obsahuje pevný počet odkazů na datové bloky
    - Z offsetu v souboru lze jednoduše spočítat, který odkaz použít pro přístup k datům (dobré pro náhodný přístup)
  - Několik inode se vejde do 1 bloku (velikost inode bývá např. 128 B)



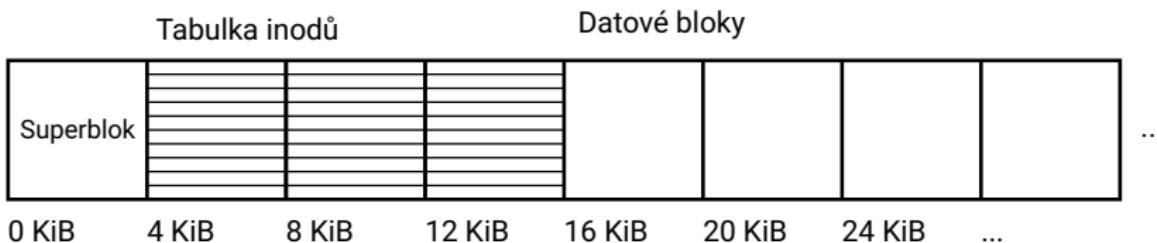
# Nepřímé bloky

- Co když je soubor větší, než počet odkazů na datové bloky v inode?
- Odkaz na další bloky nepřímo, přes blok odkazů
- Nepřímé bloky mohou být i v dalších úrovních
- I u nepřímých bloků lze jednoduše spočítat, který odkaz použít pro přístup k datům (n-ární vyhledávací strom – dobré pro náhodný přístup)



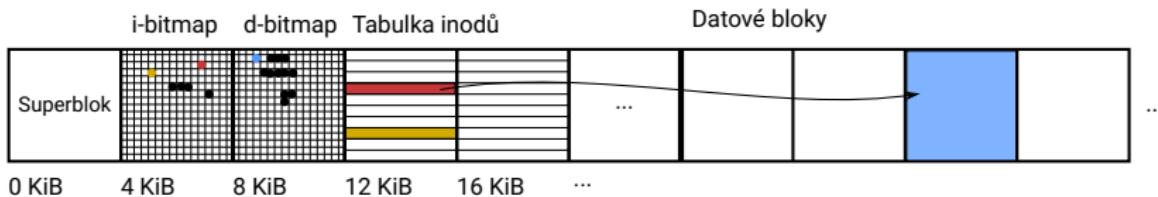
## Rozložení na disku

- Pevný počet inodů
  - inode lze nalézt na základě jeho indexu v tabulce
  - inode je zkratka *index node*
  - Superblok – informace o souborovém systému
    - celková délka, počet inode, ...
    - počet volných bloků a inode
    - odkaz na záložní kopii superbloku
  - Kořenový adresář: např. v inode č. 0



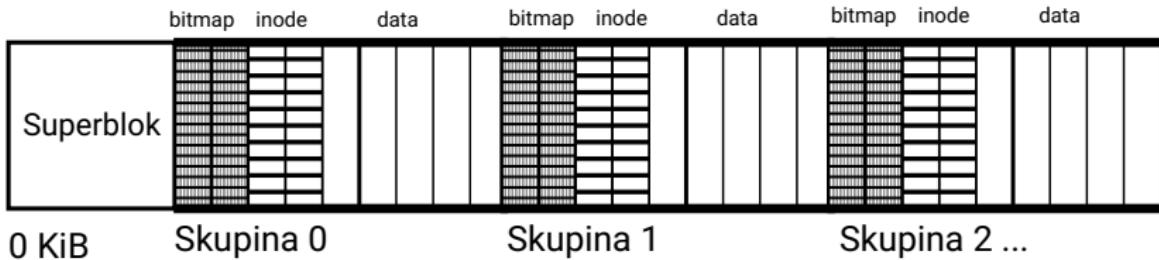
# Hledání volného místa

- Jak poznat, který inode je volný (např. při vytváření nového souboru)?
  - Např. sekvenčním procházením všech inode (neefektivní, ale fungovalo by to)
- Jak poznat, který datový blok je volný?
  - Těžko – i blok plný nul může být platným obsahem souboru
- Bitové mapy pro inode a datové bloky
  - každý bit udává obsazenost inodu nebo datového bloku



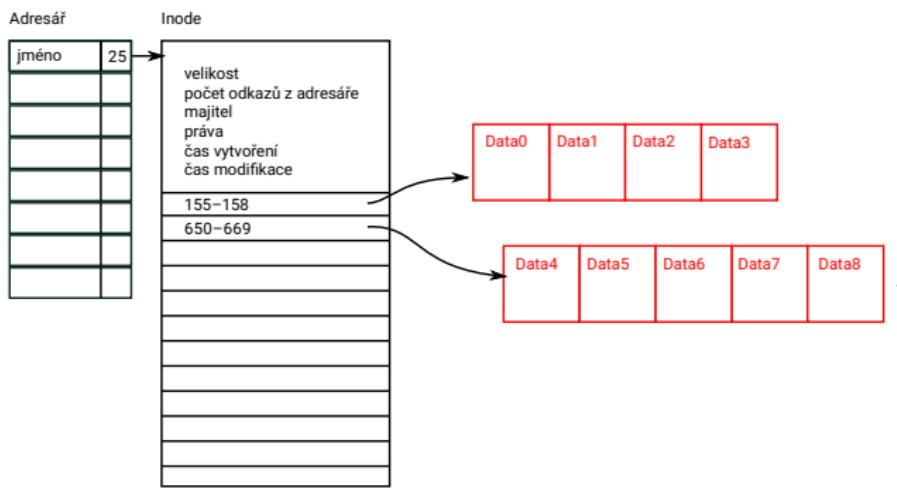
## Skupiny (ext2-4)

- Při práci se souborem je potřeba pracovat s bitmapou, inodem a datovými bloky
  - Disky (zejména rotační, ale částečně i SSD) přistupují rychleji k blokům uložených blízko sebe
  - Co když datové bloky budou až na konci disku?
    - Hlavičky disků musí pořád jezdit mezi začátkem (bitmapy, tab. inode) a koncem disku (data)
  - Řešení: skupiny
    - Souborový systém se snaží alokovat datové bloky ve stejné skupině jako inode souboru

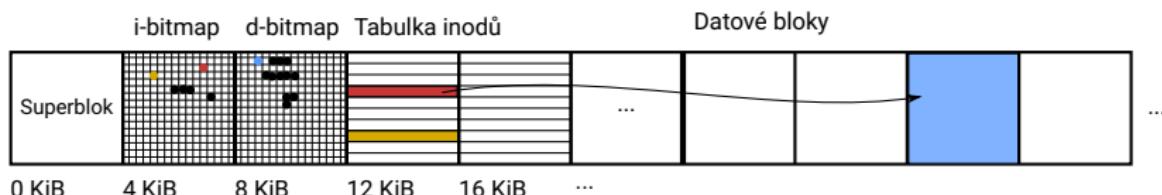


# Extents

- Tabulky bloků nejsou efektivní pro velké soubory, velká režie
- Moderní souborové systému mohou odkazovat místo na jednotlivé bloky na celé souvislé skupiny bloků
- Odkazovaná skupina s více než jedním blokem se nazývá **extent**
- Implementováno v: ext4, NTFS, btrfs, ...



# Konzistence dat



- Při zápisu do souboru je potřeba měnit: bitmapy, inode/nepřímé bloky a data
- Hardware disku garantuje atomický zápis pouze jednoho sektoru
- V jakém pořadí bloky zapisovat na disk?
- Co se stane, když dojde k pádu či vypnutí systému v průběhu zapisování při následujících pořadích zápisu?
  - bitmapa, inode/nepřímé bloky, data
  - inode, data, bitmapa
  - bitmapa, data, inode
- Vždy může vzniknout v datech nějaká nekonzistence! (bude porušena integrita souborového systému)
  - Např. můžu zajistit konzistenci při vytváření či zvětšování souborů, ale zkracování souborů povede k možným nekonzistencím.

# Možná řešení problémů s integritou souborového systému

## 1 Kontrola souborového systému při startu počítače

- projdu všechny inode a nepřímé bloky
- zjistím, jestli bitmapa volných inode souhlasí se stavem tabulky inode
- zjistím, jestli bitmapa datových bloků souhlasí s informacemi v inode
- zjistím, jestli dva inode neodkazují na stejné bloky
- ...
- **Pomalé**, zejména na velkých discích!

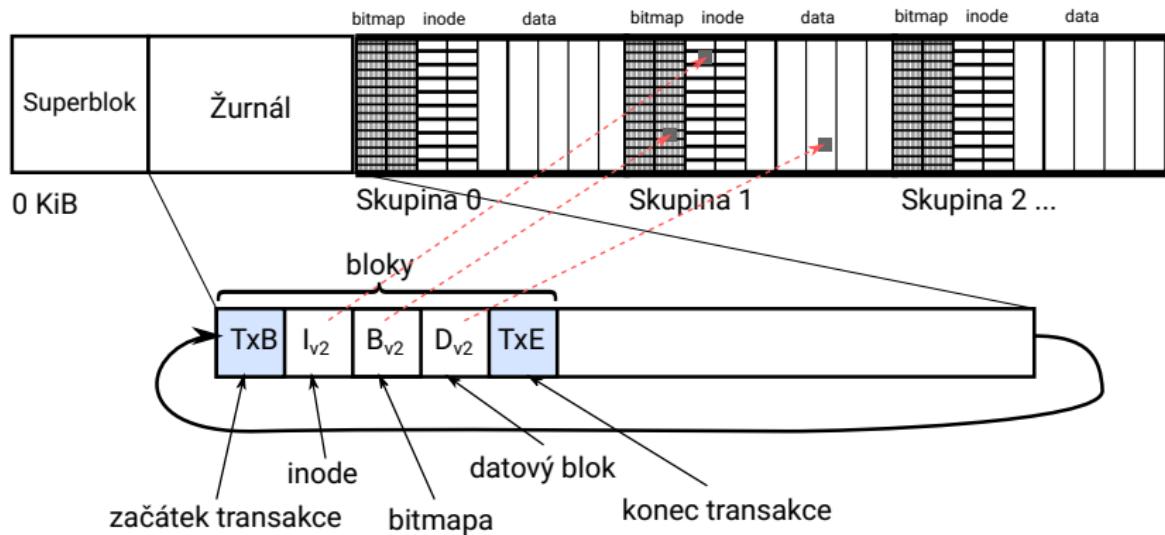
## 2 Žurnálování

## 3 Copy-on-write

# Žurnálovací systém souborů

- Před tím, než se začne souborový systém modifikovat, se uloží seznam potřebných modifikací na vyhrazené místo – **žurnál**
- Pokud dojde k pádu systému, zkонтroluje se žurnál, změny disku v něm nalezené se provedou dodatečně
- Žurnálování se někdy nazývá „dopředné logování“
- Implementováno: NTFS, ext3, ...

# Struktura žurnálovacího systému souborů (ext3)



# Bezpečný způsob změny souborového systému

## 1 Commit – zapsání transakce do žurnálu

- $TxB$ : obsahuje id transakce a čísla bloků měněného inode, bitmap a dat
- $I_{v2}$ : nová verze bloku s inode
- $B_{v2}$ : nová verze bloku bitmapy
- $D_{v2}$ : nový datový blok
- $TxE$ : id transakce, kontrolní součet

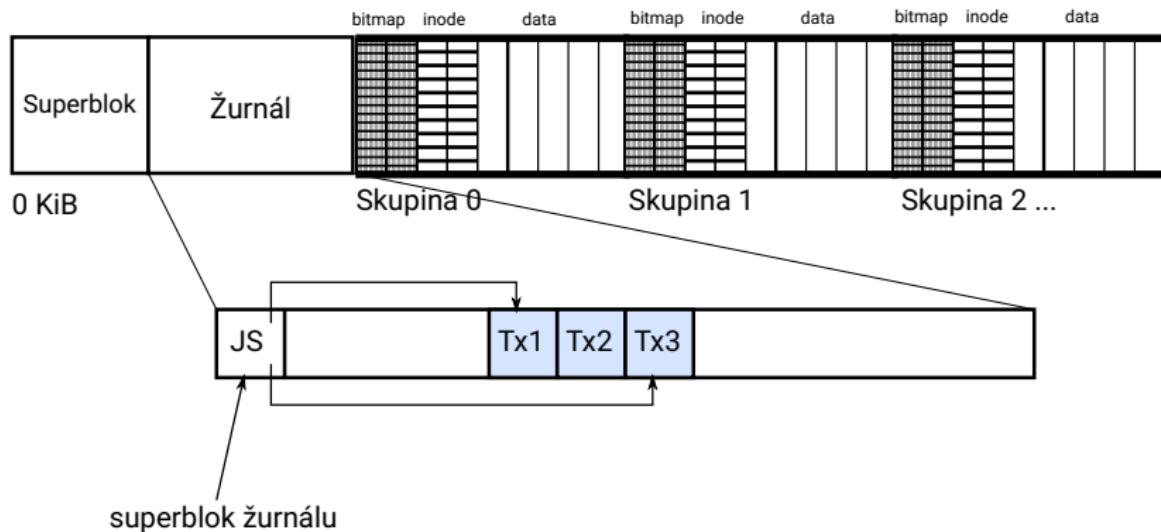
## 2 Checkpoint – provedení změn

- 1 aktualizace bloků v souborovém systému (inode, bitmapy, data)
- 2 odstranění transakce z žurnálu

# Možné scénáře pádu systému

- 1 Do žurnálu se zapíše pouze část transakce
  - Souborový systém (SS) je konzistentní a obsahuje původní data
  - Při startu OS se zjistí, že transakce v žurnálu není kompletní (viz další slide) a ignoruje se.
- 2 Do žurnálu zapíšeme celou transakci, ale neaktualizují se bloky SS
  - Při startu OS aktualizujeme bloky SS podle informací v žurnálu
- 3 Zapíšeme celou transakci, aktualizujeme bloky systému, ale neodstraníme transakci ze žurnálu
  - Při startu OS se bloky přepíší ze žurnálu – žádná změna, už zapsané byly a transakce se odstraní ze žurnálu
- 4 Zapíše se pouze část transakce (např.  $TxB$ ,  $I_{v2}$  a  $TxE$ , bez  $B_{v2}$  a  $D_{v2}$ )
  - **Problém!**
  - HW disků se snaží provádět optimalizace a může změnit pořadí vykonávání příkazů zaslaných OS
  - OS musí disku posílat speciální příkazy (tzv. bariéry), aby se data skutečně zapsala v potřebném pořadí
  - Bariéra garantuje, že příkazy zaslané před bariérou budou vykonány před příkazy zaslanými po bariéře
  - Při zápisu transakce do žurnálu se tedy disku posílá sekvence příkazů:  $TxB$ ,  $I_{v2}$ ,  $B_{v2}$ ,  $D_{v2}$ , **bariéra**,  $TxE$

# Nevyřízené transakce



- V jednom okamžiku může vypadat žurnál např. takto
- Commit transakce do žurnálu nebo její smazání se provede atomickým zápisem superbloku žurnálu

# Rychlosť žurnálu

- Pomalé
  - Commit: zápis metadat a dat do žurnálu
  - Checkpoint: aktualizace inode, bitmapy a dat podle transakce
  - Vše se zapisuje na disk dvakrát!
- Rychlejší:
  - Zapsání dat přímo do daného bloku + bariéra
  - Commit metadat: Když jsou data uložena, zapsání transakce pro změnu metadat do žurnálu
  - Checkpoint: Aktualizace inode a bitmap podle transakce
  - Jaké chyby v SS mohou nastat při pádu systému?
    - Data budou částečně aktualizována, ale ne např. velikost souboru
- Ještě rychlejší:
  - Zapsání dat přímo do daného bloku
  - Commit metadat: zapsání transakce pro změnu metadat do žurnálu
  - Checkpoint: Aktualizace inode a bitmap podle transakce (doufáme, že data zapsána také)
  - Jaké chyby v SS mohou nastat při pádu systému?
    - V souboru se mi mohou objevit náhodná data (odjinud)

Všechny módy zajišťují základní integritu SS – nekonzistentní budou maximálně soubory, se kterými se pracovalo při pádu, ne celý SS.

# Souborový systém ext4/jbd2

- Uživatel si může zvolit, jaký mód žurnálování se použije
  - **journal**: všechna data i metadata se zapisují skrze žurnál
  - **ordered** (výchozí nastavení): data se zapisují přímo, metadata skrze žurnál po zapsání dat
  - **write-back**: data se zapisují přímo, ale jejich zápis nemusí proběhnout před zápisem metadat (skrze žurnál)
- Typická velikost žurnálu: 128 MiB

# Vlastnosti Flash paměti

- Zapisovat lze pouze do vymazaného bloku
- Zapsat na jedno místo lze pouze jednou
- Mazací blok bývá mnohem větší (např. 4 MiB) než blok souborového systému (4 KiB)
- Každý blok garantuje pouze určitý počet přepsání – např. 100 tisíc

## Důsledky pro „tradiční“ souborový systém?

- Často se měnící data (např. bitmapy, či FAT tabulka) drasticky snižují životnost paměti
- Změna jednoho bytu v souboru znamená smazání a znova zapsání 4 MiB
- Garance poskytované žurnálovacím souborovým systémem neplatí pro Flash
  - Commit žurnálu musí vymazat 4 MiB data okolo commitované transakce
  - Pokud systém havaruje mezi smazáním a zápisem, přijdeme o data v žurnálu

# Řešení

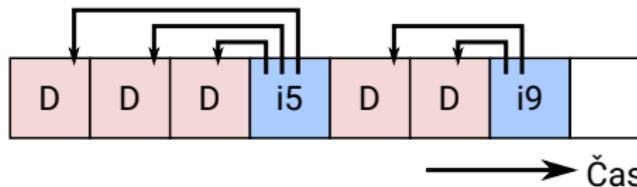
- 1 Nepoužívat Flash čipy samostatně, ale v kombinaci s řadičem, implementující „Flash Translation Layer“ (FTL)**
  - Mapuje logická čísla sektorů zaslaných OS na bloky flash pamětí tak, aby nedocházelo k nežádoucím jevům (např. neustálé přepisování stejného bloku)
  - Implementováno v SSD dicích, SD kartách, USB pamětech apod.
  - SD karty/USB paměti mají FTL často optimalizovaný pro souborový systém FAT.
    - Pokud se použije jiný souborový systém, je to pomalé a paměť dlouho nevydrží
- 2 Použít speciální souborové systémy pro Flash paměti**
  - UBIFS, JFFS2, NILFS, ...

# Protokolovací souborové systémy

## Log-Structured file systems

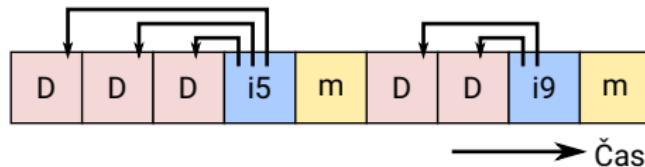
- Data se čtou převážně z vyrovnávací paměti (page cache)
- Stačí se zaměřit na operace zápisu – snaha je zapisovat data rovnoměrně po celé oblasti disku
  - Zapisuje se „od začátku do konce“ disku, starší data se nikdy nemění, ale jejich změněné kopie se zapíší na konec.
  - Zápis velkých souvislých bloků dat je velmi efektivní (není třeba znova zapisovat nezměněná data v mazacím bloku)
- Stav celého souborového systému je dán zaznamenaným protokolem událostí

Příklad: zápis dvou souborů na disk:



- Jak najdeme inody (i5, i9, ...)?
  - Sekvenčně projdeme celý disk a najdeme je. Pomalé :-)

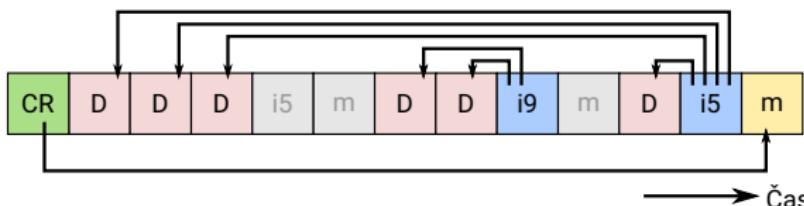
# Mapa inodů



- Abychom nemuseli při startu systému procházet můžeme si po každé změně SS uložit aktuální „mapu inodů“ (m)
- Mapa inodů obsahuje tabulku pro převod čísel inodů na čísla bloků
- Jak zjistíme, která verze mapy je poslední?

# Kontrolní region

## Check region (CR)



### Čtení souboru

- Přečti kontrolní region
- Najdi pozici mapy inodů (m)
- Najdi inode
- Přečti datové bloky

### Zápis souboru

- Zapiš datové bloky
- Zapiš změněnou kopii inode
- Zapiš změněnou kopii mapy inodů
- Aktualizuj kontrolní region

### ■ CR se pořád přepisuje – nevadí to?

- Nevadí, pokud máme např. SD kartu optimalizovanou pro FAT (viz slide 35)
- Využíváno např. F2FS (Samsung)

# B4B350SY: Operační systémy

## Grafika, GUI a HW akcelerace

Michal Sojka<sup>1</sup>



20. ledna 2021

---

<sup>1</sup>[michal.sojka@cvut.cz](mailto:michal.sojka@cvut.cz)

# Obsah I

## 1 Uživatelské rozhraní

- Smyčka událostí
- GUI knihovny

## 2 Grafický substituční OS

- HW akcelerace grafických operací
- Grafické servery

## 3 Použití GPU jako výpočetního akcelerátoru

# Uživatelské rozhraní

- Příkazová řádka (shell, ...)
- Textové rozhraní (konzolové aplikace
  - vim, top, Midnight Commander
- Grafické rozhraní (grafická okna)
  - MS Word/Libre Office, Webový prohlížeč
  - V dnešní době je velká část aplikací postavena na **webových technologiích**. Tyto aplikace nepoužívají pro zobrazování grafiky přímo služby OS, ale služby webového prohlížeče, který pak využívá služby OS.

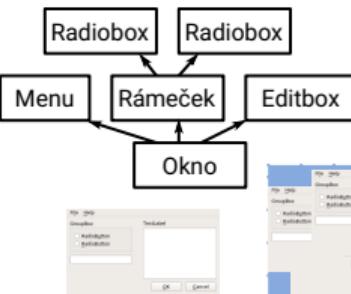
## Cíl přednášky

- Podívat se „pod pokličku“ frameworkům pro tvorbu uživatelského rozhraní a pochopit jak interagují s jádrem OS a jinými komponentami.
- Naznačit, jak fungují GPU.

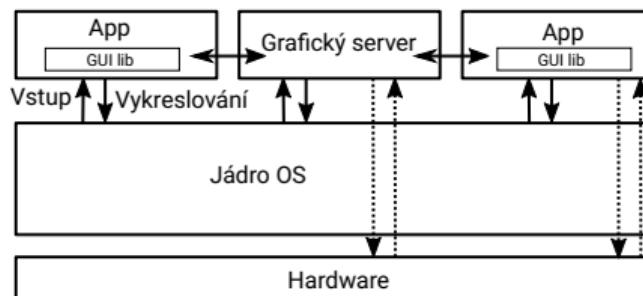
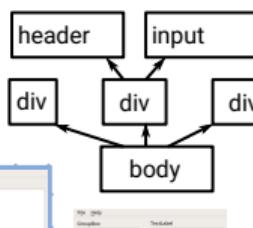
# Koncept GUI z pohledu OS

GUI = graphical user interface

Widget tree inside GUI apps



DOM tree of a web page



- Aplikace se typicky stará pouze o „své“ okno
- Kombinování oken různých aplikací má na starosti tzv. „grafický server“.
- Aplikace čte informace (události) o vstupu od uživatele
  - myš, klávesnice, touch screen
  - to má na starosti tzv. **smyčka událostí** (viz dále)
- Aplikace organzuje grafické prvky ve stromové struktuře
  - Události posílá do správných „objektů“ v aplikaci (např. aktuálně vybraný „edit box“)
  - „Objekty“ na události reagují a vykreslují se
- Vykreslování – čáry, text, obrázky, ...
  - softwarové – do framebufferu v paměti, implementováno knihovnou, kromě finálního zobrazení není potřeba OS
  - HW akcelerované – příkazy k vykreslování se posílají do GPU (zpravidla za pomoci jádra OS)

# Smyčka událostí

## Event loop

- Typicky v hlavním vlákně aplikace:

```
while (!quit) {  
    wait_for_event // keyboard, mouse, timer, ...  
    handle_event   // send to focused widget for handling  
}
```

- Hlavní smyčku většinou nepíše programátor, ale nachází se uvnitř GUI frameworku (např. `app.run()`)
- U vícevláknových aplikací může být smyčka událostí ve více vláknech
  - Hlavní vlákno řeší například události od GUI
  - Ostatní vlákna zpracovávají události zaslané hlavním vláknem (viz níže) nebo třeba události ze sítě
- Pokud obsluha události v hlavním GUI vlákně trvá dlouho, aplikace nemůže reagovat na jiné události
  - Projevuje se to jako „zatuhlá“ aplikace, či hláška „aplikace neodpovídá“

# Dlouhotrvající obsluha události

- Dletrvající obsluha by se měla vykonávat mimo hlavní vlákno (v tzv. pracovním vlákně), aby neblokovala smyčku událostí.
- Z hlavního vlákna se pouze spustí – např. pomocí semaforu nebo zasláním události do smyčky v pracovním vlákně.
- Mnoho knihoven má omezení, že některé operace (např. vykreslování) lze volat jen z jednoho vlákna (jedná se o tzv. nereentrantní funkce)
- Proto je dokončení obsluhy v pracovním vlákně často signalizováno zasláním události zpět do hlavního vlákna (např. pomocí pipe či fronty zpráv). A teprve v hlavním vlákně se uživateli vykreslí/napiše že operace byla dokončena.

# Neblokující I/O

Linux/Unix

- Základním problémem, který musí smyčka událostí řešit je čekání na více zdrojů událostí současně (myš, klávesnice, případně síť, ...)
- Pokud aplikace otevře např. `/dev/input/mice` a zavolá `read()`, **vlákno se zablokuje** do té doby než uživatel pohne myší a na jiné vstupy (např. klávesnice) nemůže reagovat
- Je potřeba používat tzv. **neblokující I/O**
  - V Linuxu můžeme „file descriptor“ `fd` přepnout do neblokujícího režimu následovně:

```
/* set O_NONBLOCK on fd */
int flags = fcntl(fd, F_GETFL, 0);
fcntl(fd, F_SETFL, flags | O_NONBLOCK);
```

- Pokud na neblokujícím `fd` nejsou žádná data ke čtení, `read()` se okamžitě vrátí s chybou `EAGAIN` nebo `EWOUTDBLOCK`.

# Neblokující I/O – pokračování

Linux/Unix

- U neblokujícího I/O ale nechceme pořád kontrolovat, jestli na některém „file descriptoru“ (FD) nejsou připravena data ke čtení (tzv. busy waiting), protože by to zbytečně zatěžovalo procesor
- OS poskytuje systémová volání, které umí čekat na více FD najednou – např.:  
`select()`, `poll()`, `epoll_wait()`
  - Všechna dělají v principu to samé, ale mají jiné API a různou výkonnost.
  - Aplikace řekne na co všechno chce čekat a funkce pak čeká.

# poll() example

```
int retval;
struct pollfd poll_list[3] = {
    // specify which socket and events we are interested in
    { .fd = mouse_fd; .events = POLLIN },
    { .fd = kbd_fd;   .events = POLLIN },
    { .fd = sock_fd; .events = POLLIN | POLLOUT },
};

while(1) {
    retval = poll(poll_list, 3, 1000); // Wait for 3 FDs with 1000 ms timeout
    if (retval < 0) err(1, "poll"); // Print error message and exit
    else if (retval == 0) printf("timeout\n");
    else {
        if (poll_list[0].revents != 0) { /* read mouse_fd and process the data */ }
        if (poll_list[1].revents != 0) { /* read kbd_fd and process the data */ }
        if (poll_list[2].revents != 0) {
            if (poll_list[2].revents & POLLIN) { /* read data from the socket and process them */ }
            if (poll_list[2].revents & POLLOUT) { /* socket is ready to send data - we can do it now */ }
        }
    }
}
```

# Windows message loop

<https://docs.microsoft.com/en-us/windows/desktop/winmsg/using-messages-and-message-queues>

```
MSG msg;
```

```
while( (bRet = GetMessage( &msg, NULL, 0, 0 )) != 0) {  
    if (bRet == -1) {  
        // handle the error and possibly exit  
    }  
    else {  
        TranslateMessage(&msg);  
        DispatchMessage(&msg);  
    }  
}
```

- Windows poskytují vyšší úroveň – nízkoúrovňové události jsou převáděny na zprávy
- File descriptrs a pod. jsou schovány uvnitř GetMessage
- Princip je podobný

# GUI framework

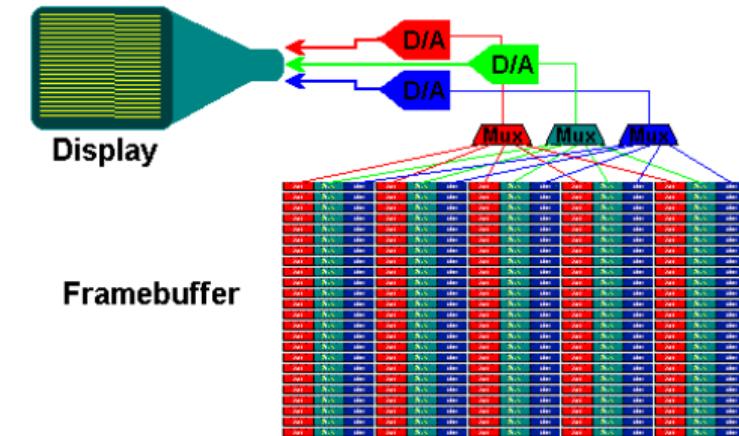
- Knihovny poskytující
  - Nezávislost na OS/HW
  - Vysokoúrovňové API (objekty, snadnost použití)
  - Základem každé knihovny je smyčka událostí
- WinForms – C#
- Qt – C++, různé OS i embedded HW
- GTK – C + podpora (bindings) jiných jazyků
- ...

# Grafický substitut OS

- Dlouhá historie
- Technologie se rychle mění
- ⇒ komplikované
  - různá API, zpětná kompatibilita, ...

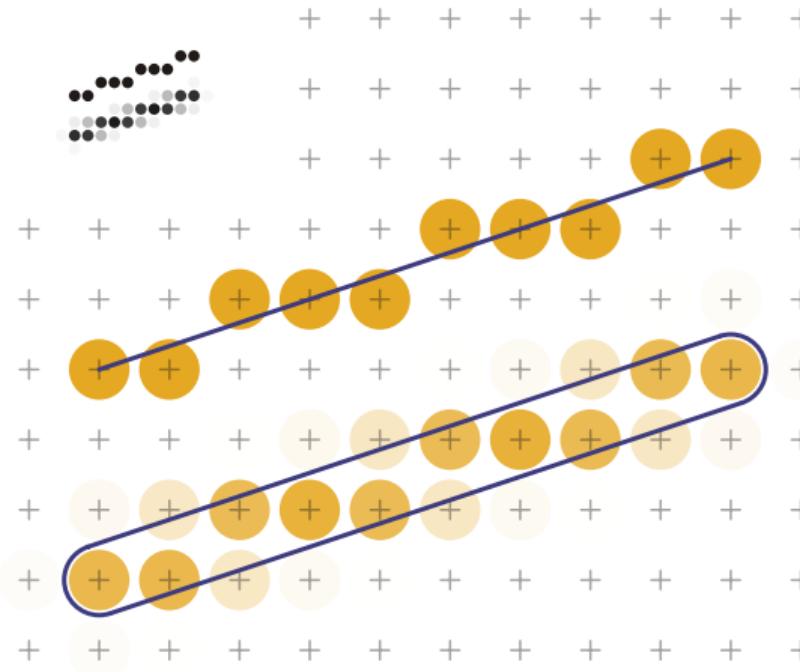
# Framebuffer (obrazový buffer)

- Starší grafické karty implementovaly pouze tzv. framebuffer
- Framebuffer je paměť jejíž obsah graf. karta převádí na signál pro displej (např. VGA)
  - dedikovaná paměť na kartě či sdílená paměť s CPU
- „Surface“ v dnešních GPU je kus paměti reprezentující např. jen jedno okno
- Vykreslování = zápis do paměti (SW)
- Dnes: low-end embedded systémy



- Formát pixelů
  - 888 RGB, 888 RGBA, 565RGB  
5551RGBA, YUV
  - ABGR 8888
  - dříve se používalo indexování (8b.) do palety

# Příklad rasterizace čáry do framebufferu



Source: Wikipedia, Phrood, CC-BY-SA 3.0

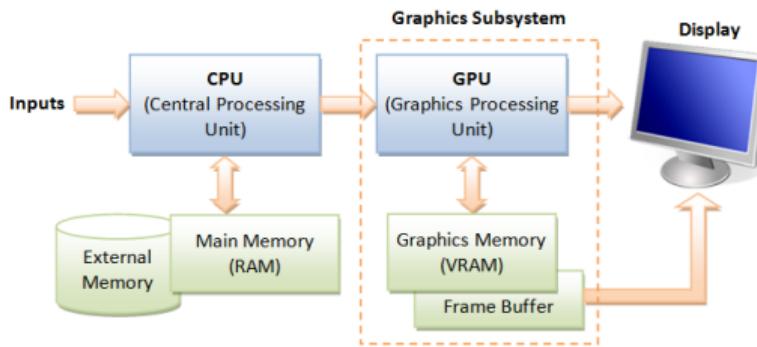
# HW akcelerace

- SW vykreslování je pomalé
  - zejména při vysokých rozlišeních
  - skládání výsledného obrazu vyžaduje mnoho „kopírování“
  - poloprůhlednost objektů vyžaduje mnoho stejných výpočtů – např. výsledná barva (pixel  $p$ ) je průměrem barev pixelů  $p_1$  a  $p_2$  dvou objektů:  $p = 0.5p_1 + 0.5p_2$
  - Anti-aliasing, ...
- Dnešní GPU je velmi výkonný paralelní počítač, který mnoho operací urychlí, nebo kompletně vykoná místo hlavního CPU
- Historicky se HW akcelerace vyvíjela:
  - 2D akcelerace
  - Pevná 3D akcelerace
  - Programovatelná 3D akcelerace
  - ...

## 2D akcelerace

- HW vykreslování kurzoru myši
  - Jeden z nejstarších typů akcelerace
  - Aby kurzor neblíkal při posunování
  - Omezená velikost (kus paměti vyhrazen pro kurzor)
  - Implementováno v poslední fázi zpracování obrazu (tzv. scanout), kdy se pixely posílají na obrazovku
- Blitter – akcelerace operací s obdélníky
  - Kopírování obdélníků
  - Výplň obdélníku konstantní barvou
  - Kopírování se roztažením/smrštěním
  - Kopírování s průhledností [Potter/Duff operátory]
- Overlay (bluescreen)
  - GPU vkládá video do místa, kde aplikace „nakreslí modrý obdélník“
- Dříve měly GPU samostatný HW pro 2D, dnes je univerzální programovatelný HW, který umí 2D i 3D
- Dnes tento typ akcelerace nabízí i malé mikrokontroléry (čipy za pár korun)

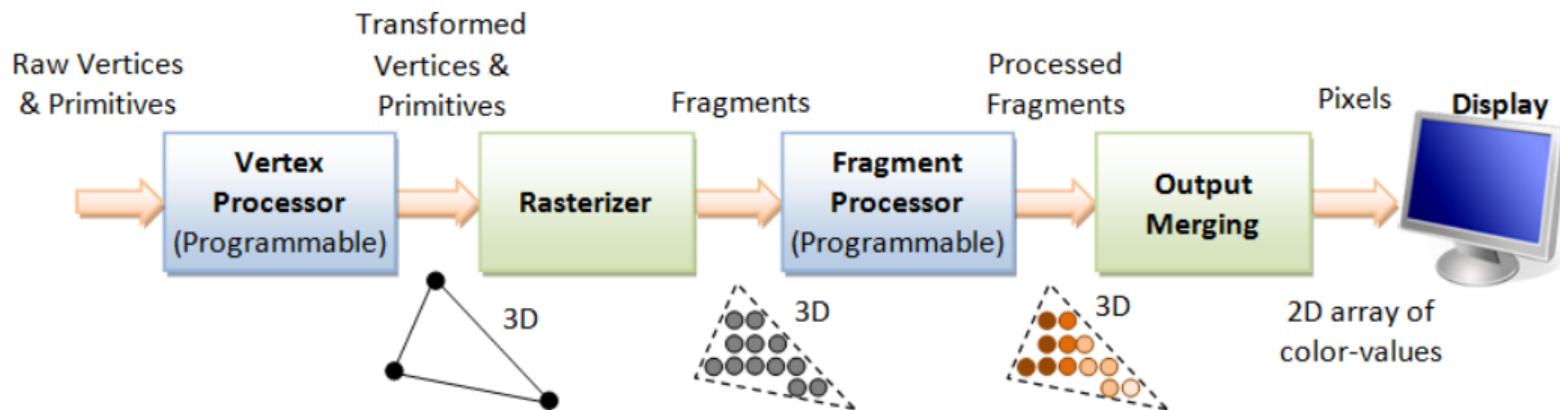
# 3D akcelerace



- Dvojité bufferování = vykreslování následujícího obrázku do „neviditelné paměti“ zatímco displej zobrazuje předchozí (hotový obrázek)

- Aplikace nezapisuje přímo do framebufferu
- Posílá příkazy a data do GPU
- GPU provádí vykreslování (zápisy do paměti) samo a paralelně na mnoha procesorech
- Výsledky ukládá buď přímo do framebufferu nebo do „neviditelné“ paměti, která je přístupná aplikacím jako „surface“

# 3D pipeline



- Vstupem je pole „vertexů“ ( $x, y, z$ ) + další informace
- Vertex procesor pracuje s vektory (rotace, posun, ...)
- Fragment procesor „obarvuje“ (stínování, textury)
- Výstupem je rastrový obrázek
- Pro práci GPU využívá mnoho paměťových oblastí
  - vstupní vertexy, výstupní rastr, hloubkovou mapu (z-buffer), ...

# Komunikace s GPU z pohledu OS

## 1 Paměťově přístupné I/O (MMIO)

- Registry
- Část paměti na GPU

## 2 DMA

- Kopírování dat
- Fronta příkazů

## 3 Přerušení

- dokončení operace

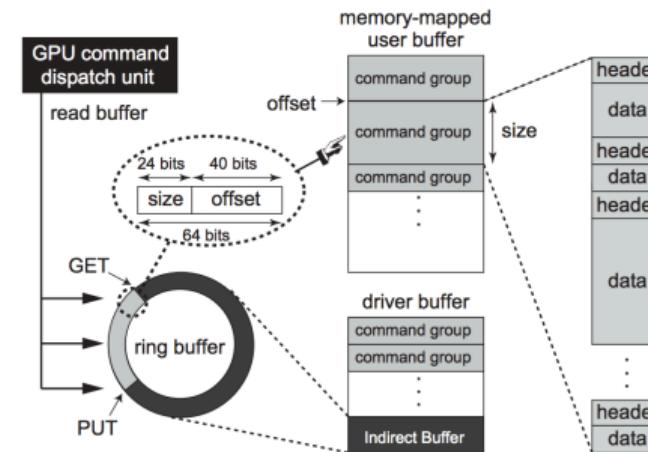


Figure 4. GPU command submissions.

- Příkazy pro GPU si připraví aplikace ve spolupráci s knihovnami (např. libGL.so)
- Knihovna je závislá na HW
- Ovladač GPU (v jádru OS) se předá ukazatel na příkazy pro GPU
  - Ovladač provede bezpečnostní a jiné kontroly a vloží novou položku do kruhového bufferu
- GPU čte příkazy z kruhového bufferu, vykonává je a výsledky zapisuje do paměti

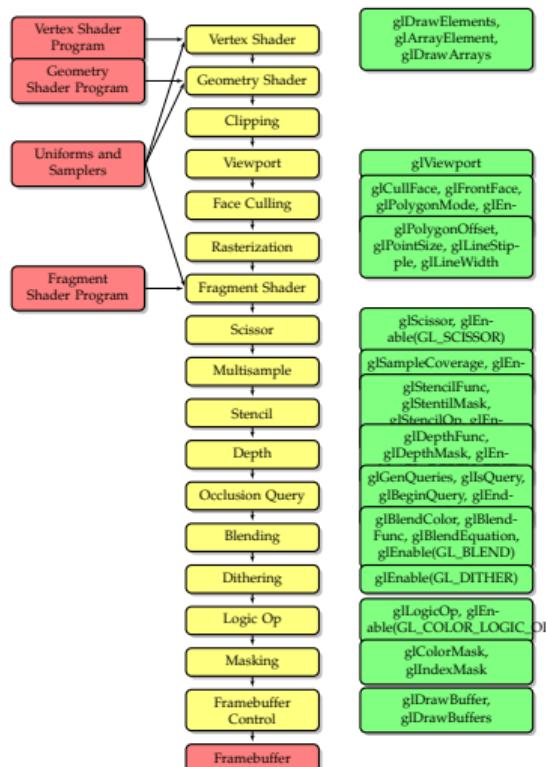
## DRI/DRM (Linux)

- Direct Rendering Interface/Direct Rendering Manager
- Dovoluje více aplikacím současný přímý přístup ke GPU (skrze knihovny)
- Obsahuje paměťový alokátor pro paměť na GPU
- Řeší koherenci paměti mezi CPU–GPU
  - Nutnost explicitně vyprázdnit cache po dokončení operací apod.

# (De)kódování videa

- Probíhá v několika fázích
- Dekódování
  - 1 Dekomprese („unzip“)
  - 2 Inverzní diskrétní kosinová transformace
  - 3 Kompenzace pohybu
  - 4 Převod barevného prostoru (YUV → RGB)
  - 5 Zvětšování/zmenšování

# OpenGL



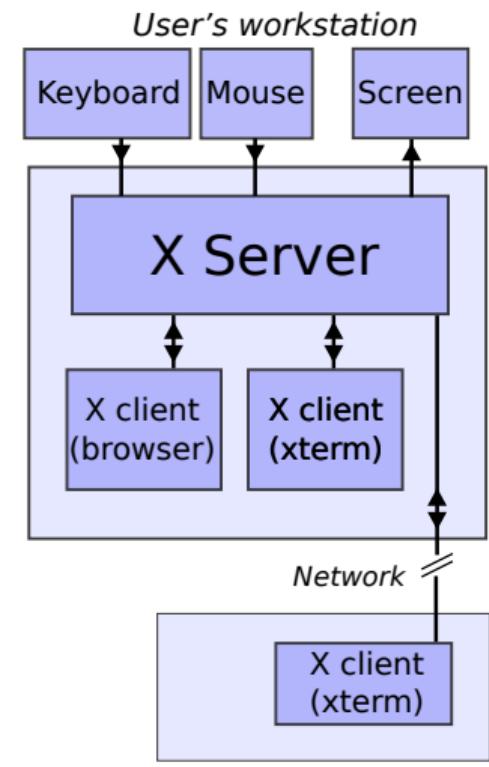
- Specifikace definující platformě nezávislé API pro grafické operace
- Lze implementovat pomocí SW renderování nebo HW renderování
- Vykreslovací pipeline je složitější než na předchozím obrázku

# Grafické servery

- X server (Unix)
- Kompozitory

# X server – grafický server pro UNIX

- Privilegovaná aplikace, umožňující ostatním aplikacím grafický výstup a čtení událostí
- Komunikace pomocí protokolu (socket)
- Síťová transparentnost
- Mimo jiné implementuje i schránku (clipboard) apod.
- Dnes
  - Aplikace nevykreslují pomocí komunikace s X serverem, ale pomocí komunikace s GPU
  - Dnes je X server „jen“ obálka implementující clipboard a kombinující okna aplikací dohromady (nadsázka)

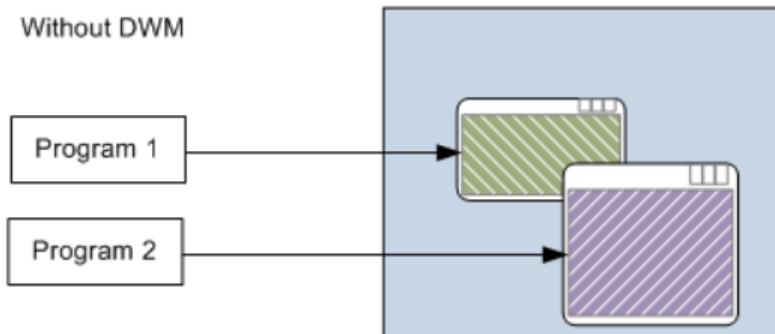


## Grafický kompozitor

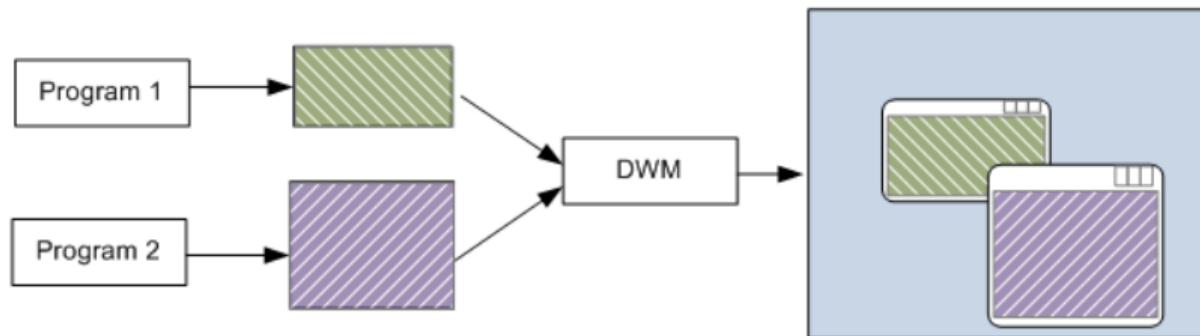
- V Linuxu např. Wayland, ve Windows od Windows Vista (DWM)
- Aplikace renderují své okénko samy pomocí přímé komunikace s GPU
- Výsledný „surface“ předají do kompozitoru
- Kompozitor přidá rámečky, stíny, průhlednost, animace, atd. a vytvoří výslednou podobu celé obrazovky (rovněž pomocí GPU)
- Při komunikaci mezi aplikací a kompozitorem se nemusí „surface“ kopírovat – „surface“ spravuje jádro a aplikace si předávají se jen odkazy (např. file descriptor)
  - Musí být zajištěna bezpečnost, aby neoprávněné aplikace nemohly vidět/modifikovat okna jiných aplikací.

# Grafický kompozitor

Without DWM



With DWM

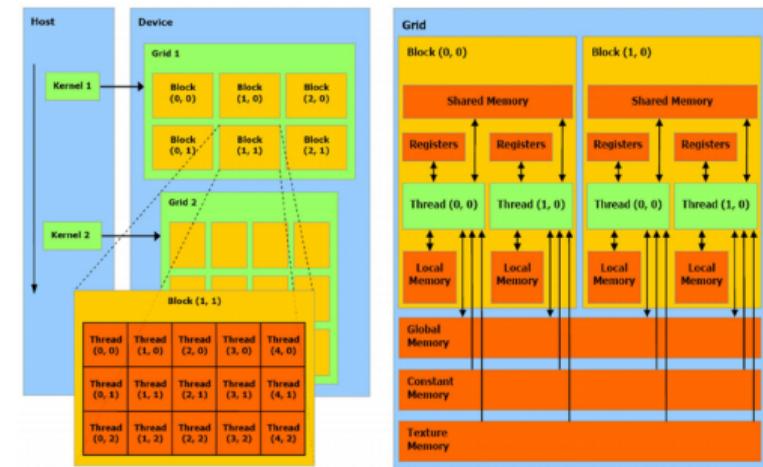


# HW architektura dnešních GPU



# Paměťová hierarchie

- Různé adresové prostory
- MMU jednotka není jen v CPU, ale i v GPU
  - Adresy: CPU virtuální, CPU fyzické, GPU virtuální, GPU fyzické
- GPU paměť není koherentní s CPU



# Psaní paralelních programů

- Většinou jako rozšíření jazyků C/C++
- Pro Akcelerátory
  - CUDA
  - OpenMP
  - OpenCL
- Pro multi-core CPU
  - OpenMP
  - TBB
  - Cilk Plus

# CUDA

- Nízkoúrovňové API od firmy NVIDIA
- Kernel – funkce, která je vykonávána paralelně na akcelerátoru

```
#include <stdio.h>

__global__
void saxpy(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

int main(void)
{
    int N = 1<<20;
    float *x, *y, *d_x, *d_y;
    x = (float*)malloc(N*sizeof(float));
    y = (float*)malloc(N*sizeof(float));

    cudaMalloc(&d_x, N*sizeof(float));
    cudaMalloc(&d_y, N*sizeof(float));

    for (int i = 0; i < N; i++) {
        x[i] = 1.0f;
        y[i] = 2.0f;
    }

    cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

    // Perform SAXPY on 1M elements
    saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);

    cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);

    float maxError = 0.0f;
    for (int i = 0; i < N; i++)
        maxError = max(maxError, abs(y[i]-4.0f));
    printf("Max error: %f\n", maxError);

    cudaFree(d_x);
    cudaFree(d_y);
    free(x);
    free(y);
}
```

# OpenMP

- Původně pro vyvíjeno paralelní CPU (multi-core)
- Později přidána i podpora speciálních akcelerátorů jako GPU pomocí *#pragma omp target...*
- Anotace pomocí direktiv

```
int main(int argc, char **argv)
{
    int a[100000];

#pragma omp parallel for
    for (int i = 0; i < 100000; i++) {
        a[i] = 2 * i;
    }

    return 0;
}
```

# OpenCL

```
// Multiplies A*x, leaving the result in y.  
// A is a row-major matrix, meaning the (i,j) element is at A[i*ncols+j].  
__kernel void matvec(__global const float *A, __global const float *x,  
                      uint ncols, __global float *y)  
{  
    size_t i = get_global_id(0);           // Global id, used as the row index.  
    __global float const *a = &A[i*ncols]; // Pointer to the i'th row.  
    float sum = 0.f;                     // Accumulator for dot product.  
    for (size_t j = 0; j < ncols; j++) {  
        sum += a[j] * x[j];  
    }  
    y[i] = sum;  
}
```

# B4B350SY: Operační systémy

## Virtualizace

Michal Sojka  
[michal.sojka@cvut.cz](mailto:michal.sojka@cvut.cz)



December 10, 2020

# Obsah

## 1 Úvod

## 2 Virtualizace celého systému

- Softwarová virtualizace CPU
- Hardwarově asistovaná virtualizace
- Virtualizace vstupu a výstupu

## 3 Kontejnery

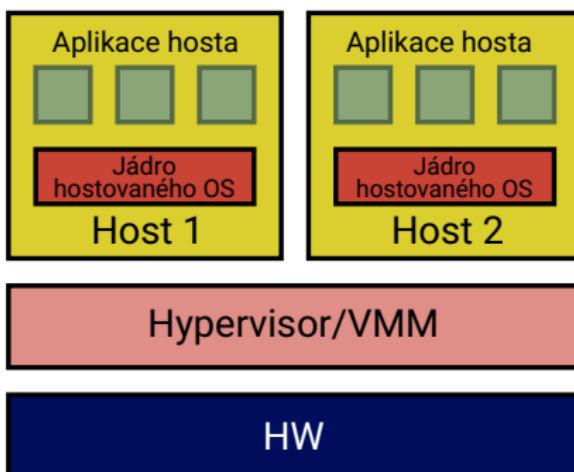
# Co je virtualizace?

- Základní myšlenka: abstrakce hardwaru počítače a jeho částečná emulace v SW
  - Operační systém, který běžel na fyzickém hardwaru běží na virtuálním HW (**virtuální stroj, VM**)
- Hlavní komponenty:
  - **Hostitel** (host) – fyzický hardware na kterém vše běží
  - **Hypervizor a virtual machine monitor (VMM)** – SW implementující virtuální hardware
  - **Host** (guest) – SW (typicky OS) běžící na virtuálním HW
- Jeden hostitel typicky může hostit více hostů najednou
- Používaná od 70. let, zejména na mainframech firmy IBM
  - Popek a Goldberg definovali požadavky pro virtualizaci počítačové architektury v r. 1974
  - Architektura x86 je plně virtualizovatelná od r. 2005

# Typy hypervizorů

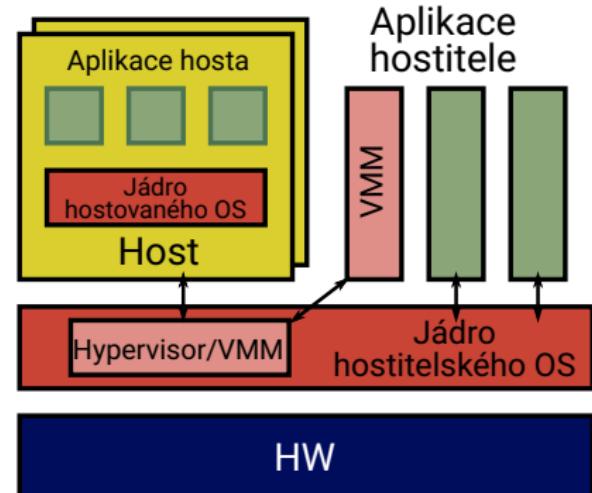
## Typ 1 – nativní

- Příklady: VMware ESX, Citrix XenServer, NOVA



## Typ 2 – hostovaný

- Příklady: Oracle VirtualBox, VMware Workstation, Parallels Desktop, Qemu



- Hranice mezi typy není vždy zřejmá. I mnohé hypervizory typu 1 mohou spouštět nativní aplikace – např. konfigurační nástroje.

# Výhody virtualizace

- **Izolace** virtuálních strojů (VM) mezi sebou
  - V ideálním případě se útok či virus nerozšíří z jednoho VM do ostatních
- **Nezávislost** softwaru na hardwaru konkrétního počítače
- Běh **více různých OS** na jednom počítači
- Možnost **pozastavení** běhu VM
  - Pozastavenou VM lze zkopirovat či přesunout na jiný počítač a **pokračovat** v běhu jinde.
- Možnost **živé migrace** – běžící VM je přesunut na jiného hostitele bez přerušení přístupu uživatelem
- Vytváření **šablon**
  - šablona = obraz OS s aplikací, distribuce všeho dohromady zákazníkům, možnost spustit víckrát
- Při **vývoji OS** „nepadá“ celý počítač
  - Lze použít pro vývoj částí OS nezávislých na HW
  - Pro vývoj ovladačů většinou nevhodné
  - Reverzní inženýrství...

# Cloud computing

- Virtualizace je základem pro tzv. „cloud computing“
- Cloud většinou umožňuje uživatelům vzdálenou správu virtuálních strojů
  - API a nástroje pro komunikaci s infrastrukturou cloudu
  - Vytváření virtuálních strojů a jejich konfigurace
- Serverless computing
  - vyšší úroveň abstrakce než „klasický“ cloud computing
  - uživatel pouze napíše software, ale nestará se kde a jak poběží
  - správu virtuálních strojů, škálování apod. řeší automaticky poskytovatel clouдовých služeb

# Je virtualizace skutečně potřeba?

- Hlavní výhody virtualizace ze strany 6 jsou zároveň vlastnosti požadované od každého OS:
  - OS izoluje aplikace (procesy) mezi sebou
  - Aplikace jsou díky OS nezávislé na HW
- Dnešní popularita virtualizace je důsledek nedokonalosti běžných OS
  - Kdyby byl OS dokonalý, nepotřebujeme spouštět více OS
  - ...

## Základní pravidlo softwarového inženýrství

Každý problém softwarového inženýrství lze vyřešit přidáním vrstvy abstrakce (layer of indirection). Jedinou výjimkou je problém příliš mnoha vrstev abstrakce.

—David J. Wheeler

# Proč jsou běžné OS nedokonalé a co s příliš mnoha vrstvami abstrakce?

- V minulosti byla preferována rychlosť před bezpečností (izolací)
  - ⇒ monolitická jádra OS: stačí jediná zranitelnost a celý systém je kompromitován
- OS jsou složité
  - Změna architektury OS by byla komplikovaná a nefungovaly by staré aplikace
  - Bylo jednoduší emulovat (virtualizovat) HW než hledat řešení pomocí změn (uzavřených) OS

## Možná řešení

- 1 Jedné vrstvy abstrakce se zbavíme jejím přesunem do hardwaru:  
HW akcelerace virtualizace ⇒ lepší HW (viz dále)
- 2 Lepší architektura OS – mikrojádro (začínají se prosazovat v mnoha aplikacích, zejména pokud jde o bezpečnost)

# Typy virtualizace

- **Virtualizace celého systému** – hostovaný systém neví, že běží na virtualizovaném systému (viz dále)
- **Paravirtualizace** – virtualizovaný systém ví, že neběží na skutečném HW a „dobrovolně“ spolupracuje s hypervizorem (tj. explicitně volá jeho služby)
- **Emulace celého systému** – vše je emulováno v SW, včetně vykonávání instrukcí. Např. Qemu umí vykonávat programy pro ARM na x86.
- **Virtualizace běhového prostředí programu** – Java VM, C# VM (mimo rámec tohoto předmětu)
- **Kontejnerizace aplikací** – viz dále

# Virtualizace CPU

- Klasické CPU vykonává kód ve dvou režimech (módech):
  - **uživatelském** (uživatelské aplikace, x86: Ring 3)
  - **privilegovaném** (jádro OS, x86: Ring 0)
- Při virtualizaci:
  - nemůžeme nechat vykonávat hostované jádro v privilegovaném režimu – nebyla by zajištěna izolace VM mezi sebou
  - potřebujeme implementovat **virtuální uživatelský** a **virtuální privilegovaný** režim
  - **skutečný privilegovaný režim** použijeme pro hypervizor
  - uživatelský i privilegovaný virtuální režim běží ve **skutečném uživatelském režimu** procesoru (sdílí ho)
- **Důsledek:** Virtuální stroj je z pohledu OS/hypervizoru velmi podobný běžnému procesu.
- **Jak zajistit, že při systémových voláních z virtuálního uživatelského režimu přejdeme do virtuálního privilegovaného režimu a ne do skutečného privilegovaného režimu?**

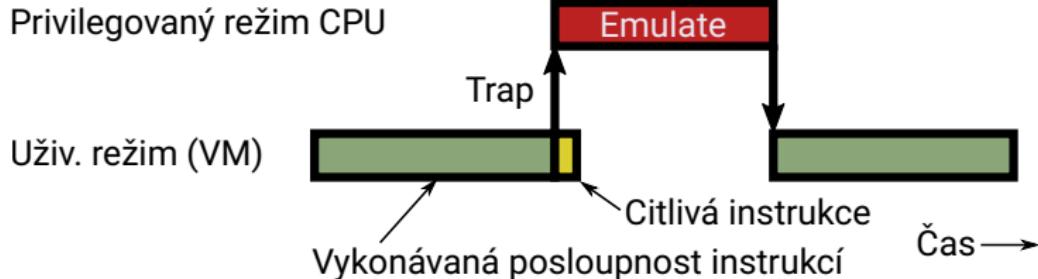
# Trap-and-emulate

## Základní princip virtualizace

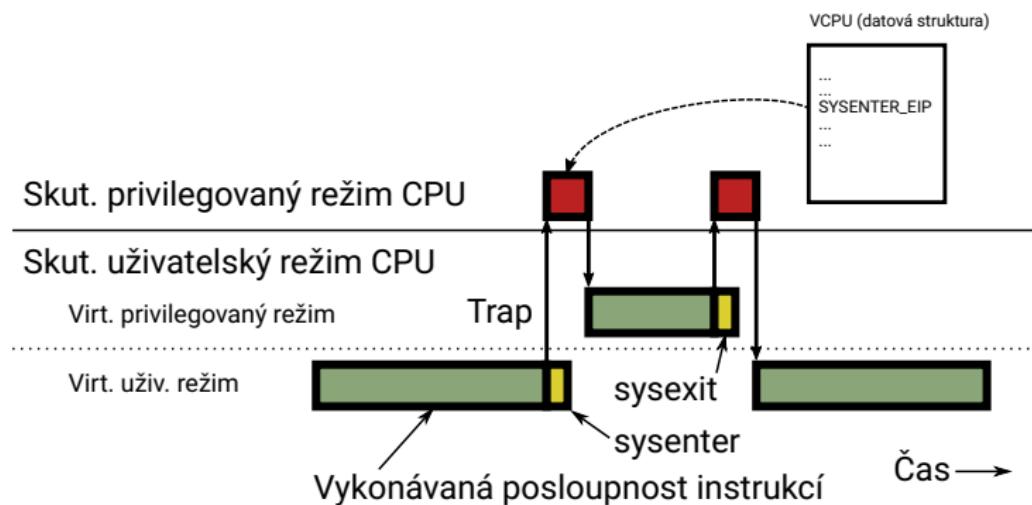
Popek a Goldberg: Požadavky na virtualizovatelnost architektury CPU

„Všechny *citlivé* instrukce musí být zároveň *privilegované* instrukce.“

- **Citlivé instrukce:** Mění *globální stav* hostitele nebo se chovají rozdílně v závislosti na *globálním stavu*.
  - Příklad: Instrukce CLI (zákaz přerušení) mění globální stav CPU.
  - Chování instrukce SYSENTER závisí na globálním stavu – přepne procesor do privilegovaného módu a skočí na **vstupní bod jádra OS** (ale každý hostovaný OS má jiný vstupní bod).
- **Privilegované instrukce:** Pokus o jejich vykonání v uživatelském módu způsobí výjimku (**trap**), která je předána do privilegovaného módu k obslužení



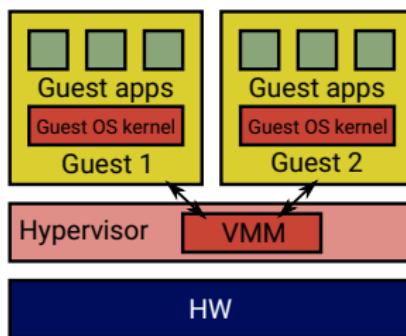
# Virtualizace systémového volání



- Uživatelský kód běží stejně rychle jako bez virtualizace
- Přechody do jádra a zpět jsou pomalejší kvůli emulaci
  - Hypervizor/VMM si přečte virtualizovaný registru SYSENTER\_EIP (položka v datové struktuře) a skočí zpátky do uživatelského režimu na přečtenou adresu.

# Rozdíl mezi hypervizorem a VMM

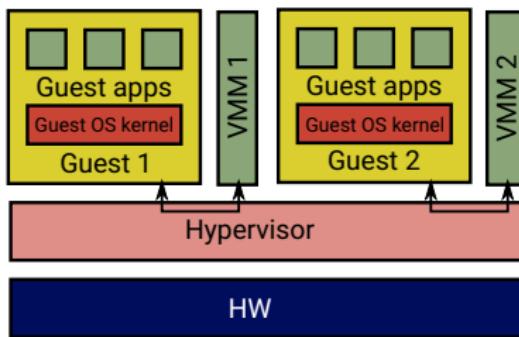
- **Hypervizor** – privilegovaný kód obsluhující výjimky generované v uživatelském režimu
- **Virtual machine monitor** – kód emulující virtuální HW (např. chování instrukce sysenter na předchozí stránce)



- Mnoho virtualizačních řešení slučuje funkci hypervizoru a VMM
- Příklad: VMware ESX, ...

# Rozdíl mezi hypervizorem a VMM

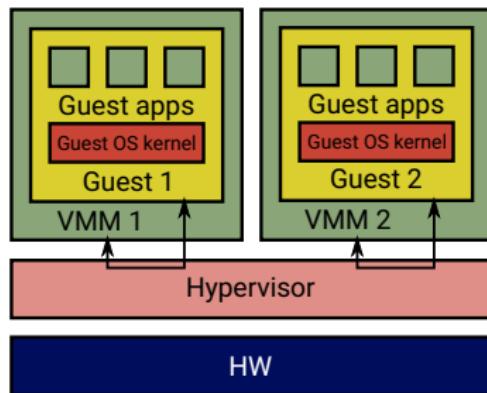
- **Hypervizor** – privilegovaný kód obsluhující výjimky generované v uživatelském režimu
- **Virtual machine monitor** – kód emulující virtuální HW (např. chování instrukce sysenter na předchozí stránce)



- Kód emulující HW je velký a složitý (zejména pro x86). Z bezpečnostního hlediska je rozumné nepouštět VMM v privilegovaném režimu.
- Hypervizor pouze odchytí výjimku a přepošle ji procesu VMM
- Příklady: NOVA, ...

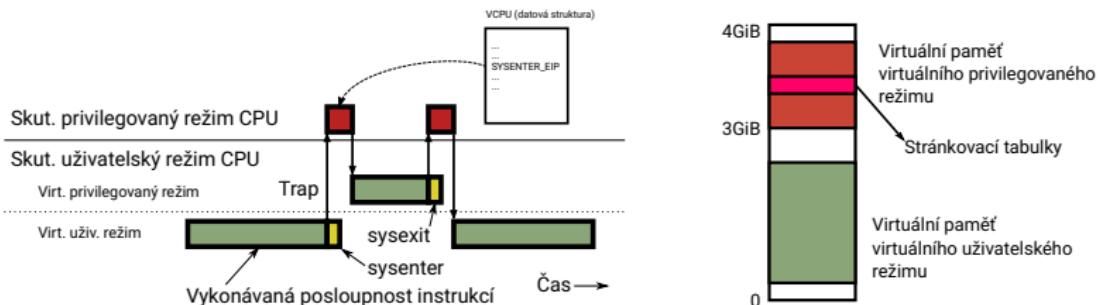
# Rozdíl mezi hypervizorem a VMM

- **Hypervizor** – privilegovaný kód obsluhující výjimky generované v uživatelském režimu
- **Virtual machine monitor** – kód emulující virtuální HW (např. chování instrukce sysenter na předchozí stránce)



- VMM a hostovaný systém často tvoří jeden proces.
- Většinou stejně bezpečné jako předchozí případ
- Příklad: KVM + Qemu, viz také příklad dále

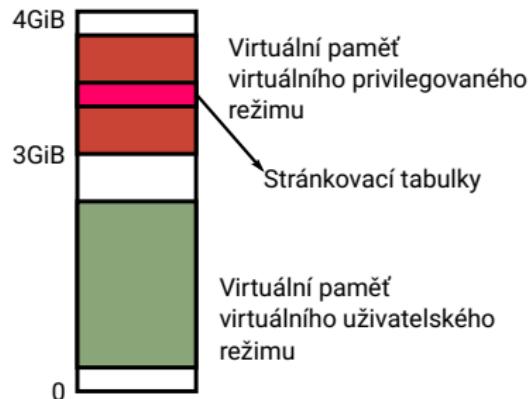
# Virtualizace jednotky správy paměti (MMU)



- Základ bezpečnosti OS je, že kód běžící v uživatelském režimu nemůže modifikovat paměť jádra
- Virtuální uživatelský a privilegovaný režim ale běží ve skutečném uživatelském režimu a tudíž mají oba stejná oprávnění.
- VMM musí emulovat jednotku správy paměti
  - Při běhu virtuálního uživatelského režimu VMM nastaví CPU, aby používalo stránkovací tabulkou, kde je povolen přístup pouze k virtuální uživatelské paměti.
  - Při běhu virtuálního privilegovaného režimu musí být přístup i do paměti hostovaného jádra – jiná stránkovací tabulka.
  - Jádro hostovaného OS nemůže mít přístup ke skutečné stránkovací tabulce.
  - Jak se dá řešit virtualizace přístupu ke stránkovacím tabulkám (např. Ptab::insert\_mapping() v OS NOVA)?

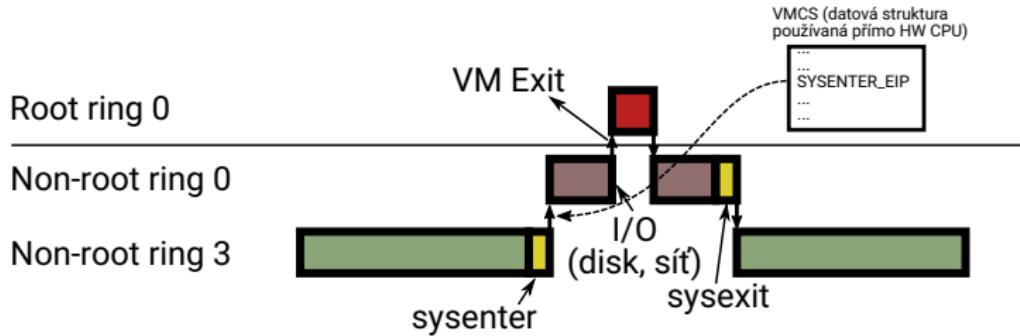
# Virtualizace stránkovacích tabulek

- Trap-and-emulate & stínové stránkovací tabulky
- Hypervizor zpřístupní hostovanému jádru paměť, kde jsou uloženy virtuální (tzv. stínové) stránkovací tabulky, pouze pro čtení
- Při pokusu o zápis do stránkovacích tabulek (`Ptab::insert_mapping`) dojde k výjimce (trap)
- VMM se podívá, jak chtěl hostovaný OS stránku nastavit a zkontroluje, jestli host nepodvádí, nedělá chybu (izolace) atd. Pokud je vše v pořádku, upraví skutečnou stránkovací tabulkou (emulate)



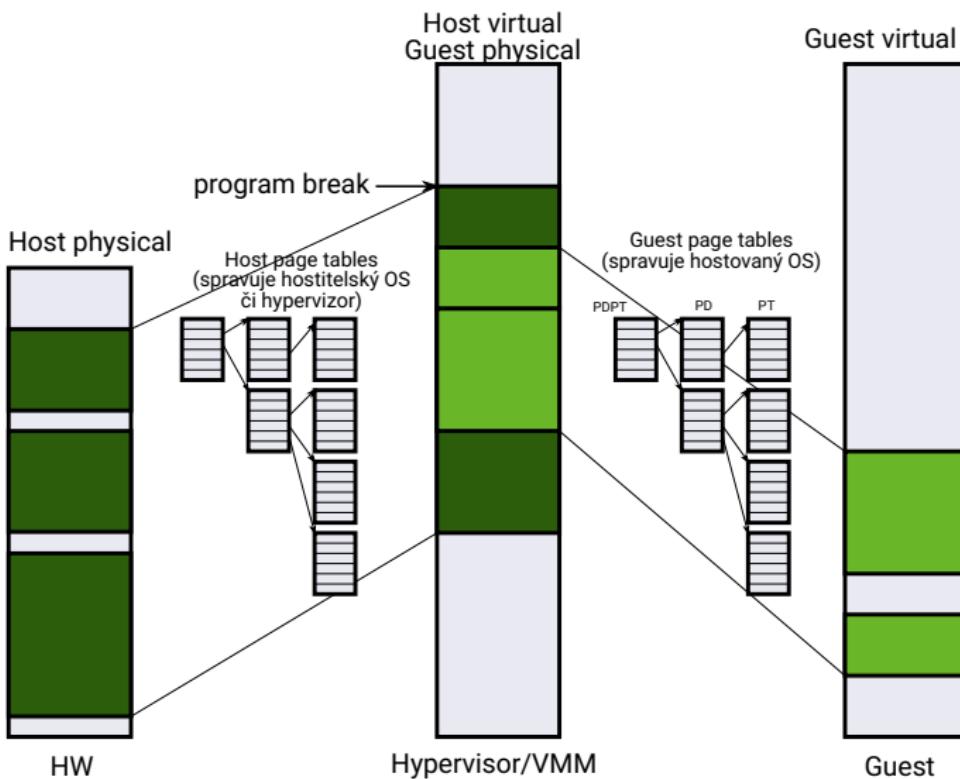
# Hardwarově asistovaná virtualizace

- Implementace Trap-and-emulate je v mnoha případech pomalá
- Moderní CPU implementují to, co typicky dělá hypervizor, přímo v HW
- Intel: VT-x
  - Zavádí nový mód procesoru: **non-root execution**
  - V tomto módu jsou všechny citlivé instrukce zároveň privilegované (v původním (tzv. root) módu to neplatí; viz např. instrukce pushf, ...)
  - Existují tedy módy: Root ring 0–3, non-root ring 0–3
  - Hypervizor/VMM může nakonfigurovat, které instrukce způsobí výjimky a přechod z non-root do root módu (tzv. VM Exit).
  - Instrukce sysenter (přechod z ring 3 do ring 0) je možné vykonat v non-root módu bez VM Exitu
  - Trap-and-emulate se používá pro emulaci I/O



# Dvouvrstvé stránkování

Odstaňuje nutnost používání stínových stránkovacích tabulek



## Dvouúrovňové stránkování

- VT-x zavádí druhou úroveň stránkování
- O jedny stránkovací tabulky se stará hostitelský OS/hypervizor
- Hostovaný OS má své stránkovací tabulky
- Překlad virtuální adresy hosta na fyzickou adresu je pomalejší
- Ale je výrazně rychlejší než SW řešení pomocí stínových stránkovacích tabulek a trap-and-emulate.

# Virtualizace vstupu a výstupu

- S většinou moderního HW se komunikuje pomocí:
  - **čtení/zápisu do registrů** (většinou mapovaných do paměti)
  - **datových struktur v paměti**
  - *Příklad (sítové rozhraní – Ethernet, zjednodušeno):* Do paměti uložím data, která chci odeslat a do registru sítového řadiče uložím adresu a délku dat. Na základě zápisu do registrů začne řadič připravená data odesílat.
- Virtualizaci paměti už máme vyřešenou
- Registry emulujeme pomocí trap-and-emulate
  - Hypervizor nastaví stránky odpovídající registrům virtuálního HW jako „not present“
  - Každý pokus hostovaného OS o přístup k registrům způsobí výjimku (výpadek stránky, page fault)
  - Hypervizor výjimku obslouží posláním adresy (a zapisovaných dat) do VMM.
  - VMM zjistí o jaký registr se jedná a provede patřičnou akci – např. odeslání Ethernetového rámce přes skutečný HW.

## Příklad: Wi-Fi rozhraní

```
$ lspci -v
02:00.0 Network controller: Intel Corporation Centrino Advanced-N 6205 [Tay
    Subsystem: Intel Corporation Centrino Advanced-N 6205 AGN
    Flags: bus master, fast devsel, latency 0, IRQ 31
    Memory at f7d00000 (64-bit, non-prefetchable) [size=8K]
    Capabilities: [c8] Power Management version 3
    Capabilities: [d0] MSI: Enable+ Count=1/1 Maskable- 64bit+
    Capabilities: [e0] Express Endpoint, MSI 00
    Capabilities: [100] Advanced Error Reporting
    Kernel driver in use: iwlwifi
```

Výpadek stránky v OS NOVA:

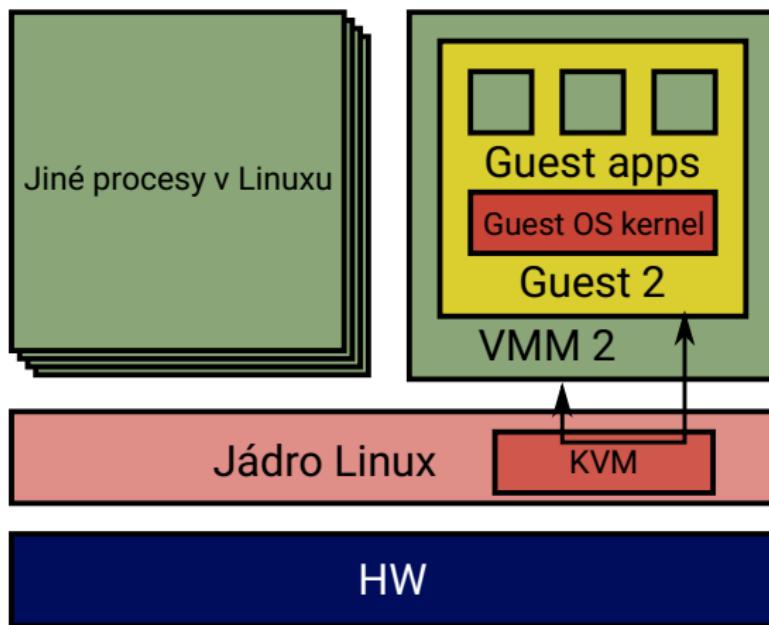
```
Ec::handle_exc Page Fault (eip=0x30f2 cr2=0x5000)
eax=0x5000 ebx=0x30a0 ecx=0x0 edx==0x0
esi=0x0 edi=0x4 ebp=0x1fe8 esp==0x1fb0
```

- Na jakou adresu se přistupovalo je uloženo v registru CPU CR2
- O jaký přístup šlo se zjistí podle instrukce na adrese EIP (mov \$123,0x5000)
- Při použití VT-x není nutné analyzovat instrukce – HW to udělá automaticky.

## Příklad: KVM

- Hostovaný hypervizor, který je součástí Linuxového jádra
- Abstrahuje hardwarově asistovanou virtualizaci pomocí API (služeb ioctl)
- Ukážeme si miniaturní VMM:
  - Nejjednodušší HW na virtualizaci: **sériový port**  
zápis 1B do registru = požadavek na odeslání daného byte
  - 1 Nastavení paměti virtuálního stroje
  - 2 Načtení kódu do paměti VM
  - 3 Spuštění kódu ve VM
  - 4 Obsluha VM Exitů a emulace sériového portu
  - 5 Goto 3
- Viz také <https://lwn.net/Articles/658511/>

## Příklad – Jednoduchý VMM pod Linuxem (KVM)



# Příklad – Jednoduchý VMM pod Linuxem (KVM)

## Inicializace

```
int kvm = CHECK(open("/dev/kvm", O_RDWR));
int vmfd = CHECK(ioctl(kvm, KVM_CREATE_VM, (unsigned long)0));

/* Allocate one aligned page of guest memory to hold the code. */
uint8_t *mem = CHECKPTR(aligned_alloc(0x1000, 0x1000));

/* Load the code to the guest memory */
int fd = open("hello.bin", O_RDONLY | O_CLOEXEC);
CHECK(read(fd, mem, 0x1000));
close(fd);

/* Map it to the second page frame (to avoid the real-mode IDT at 0). */
struct kvm_userspace_memory_region region = {
    .slot = 0,
    .guest_phys_addr = 0x1000,
    .memory_size = 0x1000,
    .userspace_addr = (uint64_t)mem,
};
CHECK(ioctl(vmfd, KVM_SET_USER_MEMORY_REGION, &region));

int vcpufd = CHECK(ioctl(vmfd, KVM_CREATE_VCPU, (unsigned long)0));

/* Map the shared kvm_run structure and following data. */
size_t mmap_size = CHECK(ioctl(kvm, KVM_GET_VCPU_MMAP_SIZE, NULL));
struct kvm_run *run = CHECKPTR(mmap(NULL, mmap_size, PROT_READ | PROT_WRITE, MAP_SHARED, vcpufd, 0));

/* Initialize CS to point at 0, via a read-modify-write of sregs. */
struct kvm_sregs sregs;
CHECK(ioctl(vcpufd, KVM_GET_SREGS, &sregs));
sregs.cs.base = 0;
sregs.cs.selector = 0;
CHECK(ioctl(vcpufd, KVM_SET_SREGS, &sregs));
```

# Příklad – Jednoduchý VMM v KVM

## Běh VM a emulace sériového portu

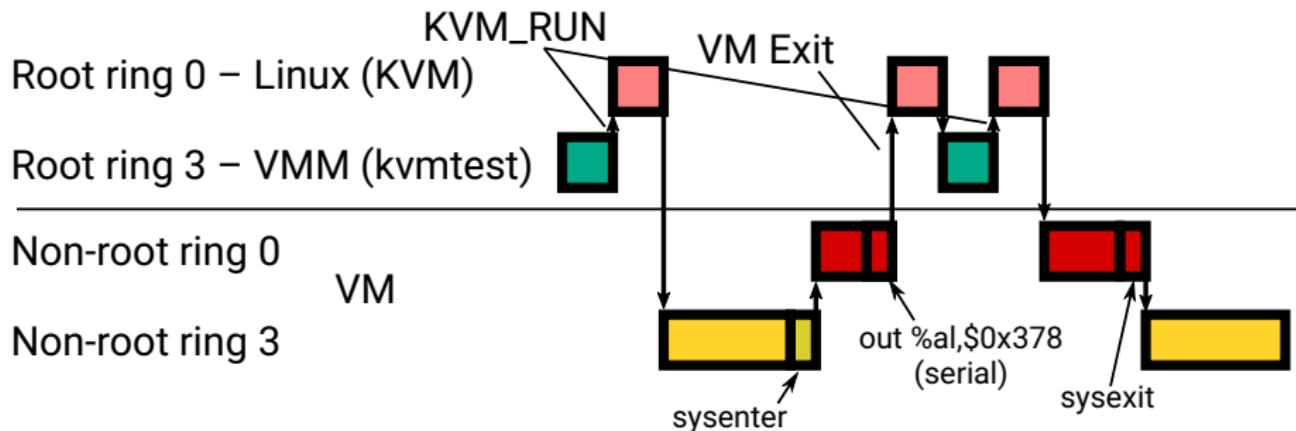
```
/* Initialize registers: instruction pointer for our code and
 * initial flags required by x86 architecture. */
struct kvm_regs regs = {
    .rip = 0x1000,
    .rflags = 0x2,
};

CHECK(ioctl(vcpufd, KVM_SET_REGS, &regs));

while (1) {
    /* Run the VM code (VM enter) */
    CHECK(ioctl(vcpufd, KVM_RUN, NULL));

    /* Handle VM exits */
    switch (run->exit_reason) {
        case KVM_EXIT_HLT:
            return 0;
        case KVM_EXIT_IO:
            if (run->io.direction == KVM_EXIT_IO_OUT && run->io.size == 1
                && run->io.port == 0x3f8 && run->io.count == 1)
                putchar(*(((char *)run) + run->io.data_offset));
            else
                errx(1, "Unhandled KVM_EXIT_IO at port %d", run->io.port);
            break;
        default:
            errx(1, "exit_reason = 0x%x", run->exit_reason);
    }
}
```

# Běh našeho VMM graficky



# Kontejnery – Motivace

- Hardwarově-asistovaná virtualizace řeší problém náročnosti virtualizace na systémové zdroje jen částečně
  - Každý virtuální stroj potřebuje paměť pro jádro OS
  - Jádro OS má nějakou režii – procesorový čas (periodické přerušení od časovače apod.)
- Často chceme pouze, aby aplikace běžící v různých VM o sobě nevěděly
- Procesy OS implicitně sdílí mnohé informace
  - Tabulka procesů – každý proces si může zjistit jaké další procesy běží v systému
  - Identifikátory uživatelů – každý proces běží s právy nějakého uživatele a umí zjistit jací další uživatelé v systému jsou.
  - Souborový systém – proces často vidí, že existují soubory jiných uživatelů i když k nim nemá přístup
- VM tyto informace nesdílí

# Kontejnery

- Řešení
  - Nepřidávat vrstvu abstrakce, ale
  - modifikovat OS, aby nebyly informace sdíleny mezi procesy
- Všechny kontejnery sdílí stejné jádro OS, ale jsou od sebe lépe izolovány.
- Linux:
  - Jmenné prostory (namespaces)
  - Řídicí skupiny (cgroups)
- Jejich použití dohromady se říká **kontejner**

# Jmenné prostory

## Namespaces

- Místo toho, aby měl každý proces přístup ke „všemu“, je jeho jmenný prostor omezen podle pravidel nastavených administrátorem
- Typy jmenných prostorů v Linuxu:
  - **Prostor identifikátorů procesů (PID)** – proces nevidí procesy v jiných jmenných prostorech
  - **Prostor identifikátorů uživatelů** – proces má např. „rootovská“ práva v kontejneru, ale ne v celém systému
  - **Prostor připojených souborových systémů** – proces vidí jen omezenou část souborového systému
  - **Prostor síťových rozhraní** – proces vidí jen podmnožinu síťových rozhraní (nebo jen virtuální síťové rozhraní). Dá se tak omezit zda/s kým bude kontejner komunikovat po síti.

# Řídicí skupiny

cgroups

- Řídí „spravedlivé“ přidělování systémových prostředků procesům
- Např. plánovač systému se snaží být spravedlivý ke všem vláknům
  - Pokud si jeden „kontejner“ vytvoří 10 000 vláken a druhý jen 10, bude první kontejner běžet 1000krát častěji než druhý
- Řešení: plánovač funguje hierarchicky
  - V první úrovni rozhoduje který kontejner poběží, v druhé úrovni které vlákno daného kontejneru.
  - Spravedlivost na všech úrovních.
- Řídicí skupiny:
  - Procesorový čas
  - Paměť (omezení na množství alokované paměti)
  - Síť (šířka pásma, latence)
  - Disky (šířka pásma, latence)
  - ...

# B4B350SY: Operační systémy

## Android

Michal Sojka<sup>1</sup>



18. prosince 2020

---

<sup>1</sup>[michal.sojka@cvut.cz](mailto:michal.sojka@cvut.cz)

# Obsah I

## 1 Úvod

## 2 Komponenty OS a Android

- Aplikace
- Souborový systém
- Init proces
  - SysV init
  - systemd
  - Android init
- Meziprocesní komunikace (IPC)
- Aplikace a frameworky (Android)

## 3 Závěr

# Mobilní OS

- Dřívější OS byly jednodušší než desktopové OS
  - Symbian OS (Nokia), Windows CE, ...
- Dnes jsou mobily výkonné jako notebooky před pár lety
  - Mobilní OS jsou upravené verze desktopových
  - Android, iOS, (Windows Mobile)
- Tato přednáška bude převážně o OS Android, což je
  - Mobilní OS od Googlu (částečně open source)
  - Linuxové jádro (trochu změněné)
  - Jiný user space než mají běžné Linuxové distribuce (Ubuntu, Fedora, ...)

# Čím se mobilní OS liší od „normálních“ OS?

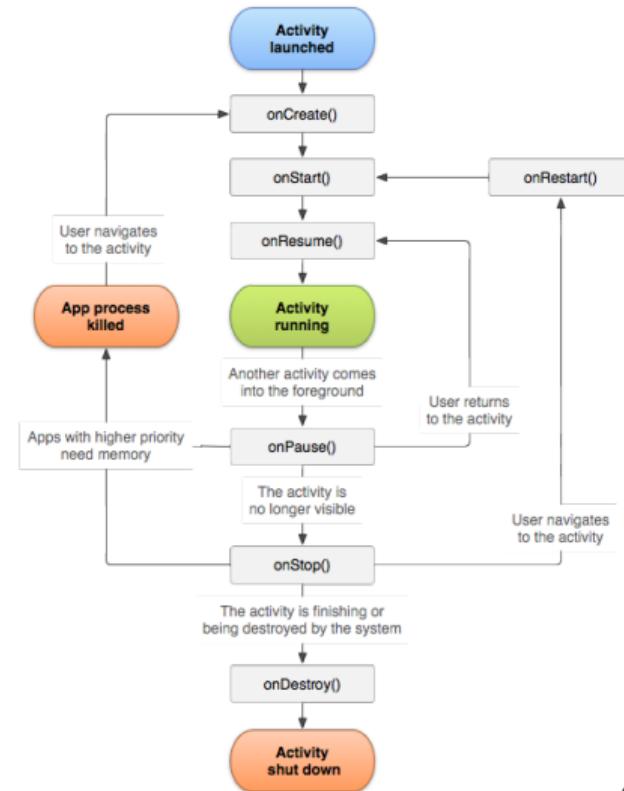
- Vše, co jsme probírali v předchozích přednáškách platí i pro mobilní OS:
  - Procesy/vlákna, synchronizace, správa paměti, IPC, ovladače, souborové systémy, grafika, ...
- To, co dělá mobilní OS mobilním jsou komponenty/knihovny/frameworky na vyšších úrovních
- Většina lidí pod pojmem „mobilní aplikace“ rozumí pouze tuto vyšší úroveň (tj. UI, design, ...)
- V této přednášce si zkusíme ukázat, jak některé vysokoúrovňové koncepty mobilních aplikací souvisí s nízkoúrovňovými záležitostmi probíranými dříve
- Podíváme se na některé komponenty či koncepty a ukážeme si, jak se liší od podobných komponent/konceptů v desktopových/serverových OS
  - Android vs. Linux na desktopu/serveru
  - Android se velmi rychle mění – ne vše, co je v této přednášce platí přesně pro poslední verze a/nebo všechny výrobce

# Aplikace

- Aplikace se skládá z
  - kódu
  - zdrojů (resources) – obrázky apod.
  - manifest – popis aplikace
  - ...
- Kód
  - Většinou vyšší programovací jazyk (Java, Kotlin)
  - Může obsahovat i nativní kód (např. C/C++) volaný skrze Java Native Interface (JNI)
- App manifest
  - Jméno aplikace + ikona + popis
  - Seznam aktivit (+ intent filters), služeb, atd. a jejich implementaci (tříd)
  - Oprávnění, která aplikace potřebuje
  - Požadavky na HW a SW (např. minimální verze Androidu)

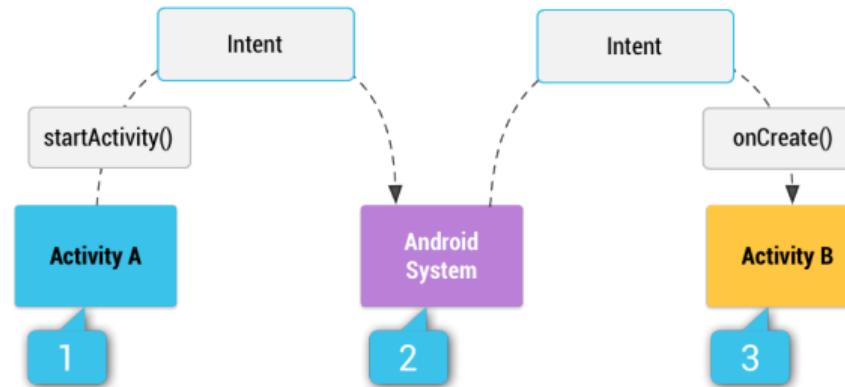
## Aktivita

- Třída reprezentující jednu obrazovku
  - Nejedná se o „imperativní“ kód, který běží od začátku do konce, ale o komponenty, které jsou volány (callback) různými frameworky (nižšími vrstvami).
  - Programátor nemá kontrolu nad tím, kdy bude proces aplikace spuštěn a ukončen
    - Např. když je málo paměti
  - Aktivity se spouští/přepínají tzv. **intenty**, což je forma meziprocesní komunikace



# Intent

- Žádost o provedení akce v jiné komponentě
  - Např. spuštění aktivity
- Explicitní – je přesně řečeno, která aplikace akci provede
- Implicitní – uživatel si může vybrat aplikaci (např. otevření webové stránky)



# Linux/UNIX

## Filesystem Hierarchy Standard (FHS)

- Specifikuje hierarchii a obsah adresářů Linuxových distribucí
- Jedna aplikace je „rozprostřena“ do mnoha různých adresářů (s výjimkou /opt)
  - **/bin** – programy, UNIXové příkazy (dnes často symlink do /usr/bin)
  - **/boot** – soubory potřebné pro boot systému (jádro, initramfs)
  - **/dev** – pseudosoubory pro komunikaci s ovladači
  - **/etc** – konfigurace systému a jednotlivých programů
  - **/home** – domovské adresáře uživatelů
  - **/lib** – knihovny
  - **/media** – připojná místa pro externí média (USB flash, CDROM, ...)
  - **/mnt** – dočasné připojené souborové systémy (např. síťové)
  - **/opt** – adresáře pro aplikační software – co adresář, to aplikace
  - **/proc** – virtuální souborový systém a informacemi o procesech a jádru
  - **/root** – domovský adresář správce systému
  - **/run** – RAM disk pro běhová data (zmizí po vypnutí systému)
  - **/sbin** – programy pro správce systému
  - **/srv** – data poskytovaná daným systémem (např. webovými servery)
  - **/sys** – virtuální souborový systém s informacemi o zařízeních apod.
  - **/tmp** – adresář pro dočasné soubory
  - **/usr** – podobná hierarchie jako v „/“ pro read-only data
  - **/var** – proměnné soubory (logy, mailboxy, data programů /var/lib, ...)

# Android

## Partitions

- **boot** – obsahuje jádro a počáteční RAM disk (initramfs)
- **cache** – cache pro stáhování aktualizací systému
- **recovery** – jádro a jiný initramfs pro obnovu systému
- **system** – `/system`
- **userdata** – `/data`

## Souborový systém

- Je tvořen initramfs do kterého jsou připojeny (mount) adresáře z flash
  - **/init** – init proces (viz dále)
  - **/sbin** – kritické programy jako např *adb*, *healthd* a *recovery*
- Připojené adresáře
  - striktní oddělení systému a dat
  - **/system** – systémové komponenty Androidu – read-only
  - **/data** – uživatelská data
    - nejsou přemazána když se aktualizuje systém
    - lze je šifrovat

# Android – hierarchie, obsah

## /system

- **/app** – systémové aplikace (od Googlu či výrobce zařízení)
- **/bin** – nativní programy (dalvikvm, vold, ...), ladicí nástroje (adb, ...), UNIXové příkazy (cp, ls, ...), atd. ls, ...)
- **/etc** – konfigurace
- **/fonts**
- **/framework** – Javová část Androidy (.jar, .odex)
- **/lib** – nativní knihovny
- **/media** – zvuky, animace, ...
- **/priv-app** – Privileged Application
- **/usr** – Support file (keyboard mappings, ...)
- **/vendor**
- **/sbin** – další systémové programy, většinou pro ladění (strace, tcpdump, nc, ...)

## /data

- **/app** – balíky .apk instalovaných aplikací
- **/backup**
- **/dalvik-cache**
- **/data** – aplikace si tam mohou uchovávat svá data (viz níže)
- **/media** – připojená SD karta
- **/misc** – konfigurace, klíče, ...
- **/property** – uložené „vlastnosti“, které přežijí reboot
- **/user** – pro podporu více uživatelů
- **/system**
- ...

## /data/data

- **/com.android.providers.calendar** – obsahuje databases/calendar.db
- **/com.android.providers.contacts** – obsahuje databases/contacts2.db
- **/com.android.chrome** ...
- ...

## Uživatelé

- Android používá systémové „uživatele“ (UID) jinak, než běžné Linuxové distribuce
- Každá aplikace běží s právy jiného uživatele (UID tedy identifikuje aplikaci, ne uživatele)
- Tím je (mimo jiné) zajištěna ochrana dat jedné aplikace před ostatními
- Některá zařízení/verze Androidu podporují více uživatelů (lidí) – každému uživateli je přiřazeno 100000 UID (může tedy nainstalovat 100000 aplikací).

# Init proces

- V UNIXových OS je init první proces, který je spuštěn po zavedení jádra OS
  - Něco jako user/hello v naší verzi OS NOVA
- Jeho úkolem je:
  - Připojit potřebné souborové systémy
  - Spustit severy a daemony potřebné pro běh systémů
  - Spustit proces(y), které umožní lokální přihlášení uživatele (getty pro textovou konzoli, *display manager* pro grafické přihlášení)
  - Adoptovat procesy, jimž umře rodič

# SysV init

- UNIX System V (*system five*) je jedna z verzí komerčního UNIXu od AT&T (1983)
- Init proces je vytvořen spuštěním /sbin/init
- Načte /etc/inittab a vykoná, co je tam napsáno

- Runlevel = co se má spustit při bootování vypínání
- Vždy se spustí skript rcS – základní inicializace a služby systému
- Poté se spustí skript rc <N><sup>2</sup>, který spustí další služby (webový server, grafický login, ...)
- Getty (textový login) se spustí v runlevelech 2–5 a při ukončení se spustí znova

## Example (/etc/inittab:)

```

id:2:initdefault:
si::sysinit:/etc/init.d/rcS

# Runlevel 0 is halt.
# Runlevel 1 is single-user.
# Runlevels 2-5 are multi-user.
# Runlevel 6 is reboot.
10:0:wait:/etc/init.d/rc 0
#
15:5:wait:/etc/init.d/rc 5
16:6:wait:/etc/init.d/rc 6

1:2345:respawn:/sbin/getty 38400 tty1

```

# SysV init – pokračování

- respawn v *inittab* aktivuje „monitorování procesu“ a restartuje proces např. v případě nečekaného pádu
- SysV init se mnohdy často používá v případě jednoduchých „embedded“ zařízení, kdy v systému běží jen pár služeb

## Skripty rc a rcS

- Spouští ostatní služby na základě tzv. *init skriptů* (někdy také *rc skriptů*)
- Jednoduchá implementace rcS sekvenčně spouští skripty z adresáře /etc/rcS začínající na „S“ (start) nebo „K“ (kill):
 

```
for i in /etc/rcS.d/S??*; do $i start; done
```
- Příklad jmen init skriptů: S01hostname, S02udev, S15networking
- Často to jsou pouze symbolické odkazy na skripty v adresáři /etc/init.d
  - /etc/rcS.d/S15networking -> /etc/init.d/networking
  - /etc/rc0.d/K08networking -> /etc/init.d/networking

## Problémy

- Řešení závislostí mezi službami pořadím startování
- Paralelní spouštění služeb (multi-core CPU)
- Monitorování a restart havarovaných služeb
- ... vše se dá řešit pomocí různých „nadstaveb“, ale ...

# Příklad jednoduchého init skriptu

Example (S40network)

```
#!/bin/sh

mkdir -p /run/network

case "$1" in
    start)
        printf "Starting network: "
        /sbin/ifup -a
        [ $? = 0 ] && echo "OK" || echo "FAIL"
        ;;
    stop)
        printf "Stopping network: "
        /sbin/ifdown -a
        [ $? = 0 ] && echo "OK" || echo "FAIL"
        ;;
    restart|reload)
        "$0" stop
        "$0" start
        ;;
    *)
        echo "Usage: $0 {start|stop|restart}"
        exit 1
esac

exit $?
```

# systemd

- Moderní init systém pro Linux
- Řeší většinu problémů SysV init (a přináší jiné problémy)
- Umožňuje popsání závislostí mezi službami => paralelní spouštění
- Aktivace pomocí socketů – viz „Socket activation“ dále
- Snadné nastavení zabezpečení služeb a přidělování zdrojů
  - Např. omezení množství paměti a CPU pro danou službu
  - Toto implementuje linuxové jádro; systemd umožňuje pouze snadnou konfiguraci

## Watchdog

- Možnost periodické komunikace se službou
- Pokud se služba dlouho nehlásí, systemd ji restartuje

## Popis služby pro systemd (.service file)

### [Unit]

```
Description=Deferred execution scheduler  
Documentation=man:atd(8)  
After=remote-fs.target nss-user-lookup.target
```

### [Service]

```
ExecStart=/usr/sbin/atd -f  
IgnoreSIGPIPE=false  
KillMode=process  
Restart=on-failure
```

### [Install]

```
WantedBy=multi-user.target
```

# Socket activation

## ■ Co to je?

- Elegantní řešení závislostí bez nutnosti jejich explicitního popisu
- Služby jsou startovány jen/až když je někdo potřebuje

## ■ Závislosti

- Servery často poskytují své služby pomocí socketů (UNIX, TCP/localhost, ...)
- Pokud služba (proces) A potřebuje něco od služby B, připojí se k socketu služby B a pošle požadavek.
  - Může se stát, že B také potřebuje něco od A. Kterou službu spustit jako první?

## ■ Základní myšlenka:

- 1 Vytvořit sockety všech služeb (ale ne jejich procesy)
- 2 Pokud se někdo k socketu připojí, spustit proces a předat mu už „existující“ socket

## ■ Implementace:

- Služba nevytváří socket sama, ale nechá to na `systemd` (konfigurační soubor).
- Při spuštění služba „zdědí“ socket od `systemd` (`fork()`, `exec()`), který jí sdělí, který file descriptor odpovídá socketu
- Služba tedy nedostane od „systému“ jen `stdin`, `stdout` a `stderr`, ale i socket, kterým klienti posílají požadavky

# Android init

- Vzdáleně podobný SysV init
- Obsahuje navíc „System Properties“
- Místo /etc/inittab má /init.rc, /init.usb.rc/ apod.
- Nemá „runlevels“, ale umí spouštět služby na základě „triggers“ a „system properties“
  - např. při změně property se spustí/restartuje služba – podobné jako launchd v iOS)
  - Příklad změny property: Připojení k nabíječce, připojení USB, ...
- Služby jsou automaticky restartovány, pokud nejsou nakonfigurovány jako oneshot.
- Pokud je služba označena jako critical a nejde restartovat, je restartováno celé zařízení
- Podpora socket activation pro UNIX sockety
- Specifické *rc skripty*
- S jinou konfigurací funguje jako ueventd (další služba v OS Android)

## System properties

- Jsou uloženy v několika souborech (dané výrobcem /system/default.prop, persistentní /data/property/persist\*, ...)
- Přístup k properties přes /dev/socket/property\_service, kontrola přístupu podle UID, možnost mapování do paměti (mmap).
- Příklady „properties“:
  - wlan.driver.status, net.hostname, sys.boot\_completed, net.dns1, ...

# .rc soubory

## Example (init.rc – zkrácelo)

```
on boot
    ifup lo
    hostname localhost
    domainname localdomain
    write /proc/sys/net/core/xfrm_acq_expires 3600
service ueventd /system/bin/ueventd
    class core
    critical
    seclabel u:r:ueventd:s0
    shutdown critical
service console /system/bin/sh
    class core
    console
    disabled
    user shell
    group shell log readproc
    seclabel u:r:shell:s0
    setenv HOSTNAME console
on property:ro.debuggable=1
    # Give writes to anyone for the trace folder on debug builds.
    # The folder is used to store method traces.
    chmod 0773 /data/misc/trace
    # Give reads to anyone for the window trace folder on debug builds.
    chmod 0775 /data/misc/wmtrace
    start console
```

# Zygote

- Jedním z procesů spouštěných procesem `init` je tzv. zygote (uložen v `/system/bin/app_process`)
- Urychluje spouštění aplikací
- Spustí Dalvik Virtual Machine a načte všechny frameworky (třídy) OS android
- Zastaví se těsně před „načtením“ hlavní třídy aplikace, otevře `/dev/socket/zygote` a čeká na požadavky
- Přicházející požadavky obsahují jméno třídy aplikace
  - Zygote zavolá `fork()` a načte třídu aplikace
  - Fork používá mechanismus **copy-on-write**
  - Tímto způsobem se velmi rychle vytvoří proces aplikace, protože vše (JVM, frameworky, ...) už je nainicializované

# Nízkoúrovňová IPC

Obecně (v UNIXu/Linuxu)

- roura (pipe)
  - přenášení proudu dat mezi dvěma procesy jedním směrem
- UNIX socket
  - přenášení dat (proud nebo zprávy) mezi dvěma procesy (obousměrné)
  - může, ale nemusí být vidět v souborovém systému (např. /run/cups/cups.sock pro komunikaci s tiskovým serverem CUPS)
  - lze přenášet i „file descriptors“ (FD) mezi různými procesy
    - Příklad: Privilegovaný proces otevře soubor a pošle FD jinému procesu, který soubor sám otevřít nemůže.

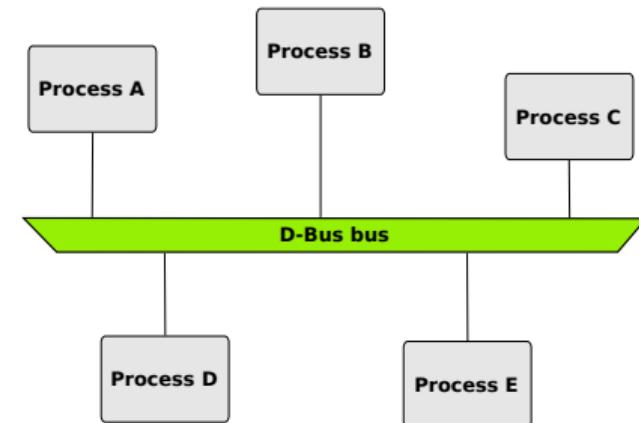
# Remote Procedure Call (RPC)

Obecně

- Možnost volat funkce/procedury ve vzdáleném procesu
- Princip:
  - 1 Při zavolání funkce se provede serializace parametrů (převod dat v paměti do formátu pro komunikaci) a odešle se žádost (data) cílovému procesu (např. pomocí socketu).
  - 2 Cílový proces data deserializuje, zjistí jakou funkci má zavolat a zavolá ji
  - 3 Pokud funkce něco vrací, výsledek se serializuje a odešle zpět.

# DBus (Desktop Linux)

- Mnoho aplikací potřebuje komunikovat na vyšší úrovni než posílání zpráv
  - Publish/subscribe
  - Komunikace jednotlivých objektů/komponent uvnitř aplikací
  - Nechce řešit, který socket použít pro danou aplikaci (v jakém procesu se nachází atd.)
  - ...
- DBus je systémový démon, který umožňuje aplikacím komunikovat na vyšší úrovni
- Aplikace si mohou definovat objekty, ptát se na objekty v jiných aplikacích, žádat o notifikace na změny v jiných aplikacích apod.



© 2015 Javier Cantero - this work is under the Creative Commons Attribution ShareAlike 4.0 license

# Android Binder

- Poskytuje RPC
- Narozdíl o DBus je implementován v jádře
- /dev/binder
- Služby:
  - 1 Hledání cílového procesu
  - 2 Přenos zpráv
    - Android Interface Definition Language (AIDL) – generuje kód, který převádí volání funkcí na komunikaci pomocí Binderu (serializace/deserializace)
    - Blokující (ioctl(BINDER\_WRITE\_READ))
  - 3 Přenos objektů
    - file descriptors
  - 4 Důvěryhodné ověření zdroje
    - adresát ví, kdo mu zprávu poslal (PID, UID)

# Dalvik VM

- Implementace Java VM od Googlu
- Aplikace se komplilují „just-in-time“ (JIT) překladačem do nativního kódu (výsledky se cachují)
- Dalvik má různé problémy – novější verze přecházejí na ART, kde se používá Ahead-of-time (AOT) komplilace

# system\_server

- Proces, kde různé systémové služby běží jako vlákna
- Podobný svchost.exe z Windows (služby jsou nahrávány z DLL knihoven)
- Psaný v Javě, služby jsou třídy v Javě
- Po inicializaci je spuštěna hlavní smyčka, která čeká na požadavky z jiných procesů a předává je službám
- Poskytované služby:
  - 1 **Bootstrap**: Installer, ActivityManager, PowerManager, DisplayManager, PackageManager, UserManager
  - 2 **Základní (Core)**: Lights, Battery, UsageStats, WebViewUpdate
  - 3 **Ostatní**: ...

# Reference

- Jonathan Levin, *Android Internals: A Confectioner's Cookbook*, Technologeeks.com, 2015, <http://newandroidbook.com/>

# B4B350SY: Operační systémy

## Free and open source software

Michal Sojka  
[michal.sojka@cvut.cz](mailto:michal.sojka@cvut.cz)



December 17, 2020

Copyright      © 2016–2020 Michal Sojka  
                  © 2014–2015 Stefano Zacchiroli ([link to the original](#))  
                  © 2004–2010 Pavel Písá, František Vacek

License      Creative Commons Attribution-ShareAlike 4.0 International License  
[http://creativecommons.org/licenses/by-sa/4.0/deed.en\\_US](http://creativecommons.org/licenses/by-sa/4.0/deed.en_US)



# Obsah

- 1 Úvod
- 2 Základy autorského práva a licence
  - Obchodní tajemství
  - Ochranné známky
  - Patenty
  - Autorské právo
- 3 Implementace FOSS pomocí autorského práva
  - Permisivní licence
  - Reciproční licence
- 4 FOSS licence
  - Výběr licence
  - Ukázky konkrétních licencí
- 5 Ekonomické aspekty FOSS
- 6 Závěr

# Počátky Free and Open Source Software (FOSS)

CZ: Svobodný a otevřený software

- 50 a 60 léta: open source (public domain) bylo normou
  - Lidé mezi sebou sdíleli kód jako kuchařské recepty nebo znalosti matematiky
- V 70. letech začaly firmy software „uzavírat“ a prodávat (AT&T Unix, Microsoft, ...)
- Richard M. Stallman (RMS)
  - 1980 – naštvala ho nemožnost opravit chybu v softwaru nové tiskárny na MIT
  - 1983 – začal vyvíjet operační systém GNU („GNU's not Unix“)
    - GNU se neujal jako samostatný systém, ale jeho komponenty jsou podstatnou součástí většiny Linuxových distribucí. Proto by se mělo říkat **GNU/Linux**.

[https://en.wikipedia.org/wiki/History\\_of\\_free\\_and\\_open-source\\_software](https://en.wikipedia.org/wiki/History_of_free_and_open-source_software)

<http://www.oreilly.com/openbook/freedom/ch01.html>

<https://www.gnu.org/gnu/rms-lisp.html>

# Svobodný software

Free software

Program je **svobodný software**, pokud uživatelé programu mají čtyři základní svobody:

- Svoboda 0: **spuštět** program k libovolnému účelu
- Svoboda 1: **studovat** jak program funguje a měnit ho
- Svoboda 2: **šířit** kopie programu a pomáhat tím ostatním
- Svoboda 3: **vylepšovat** program a **zveřejňovat** svá zlepšení

– Richard Stallman, 1986 (původní verze)

<http://www.gnu.org/philosophy/free-sw.en.html>

# Open source

- Termín vznikl okolo roku 1998 (12 let po free software)
  - Označuje otevřený vývojový model a snaží se odlišit od filozoficky a politicky motivovaného pojmu „svobodný software“
    - Open Source Initiative

<https://opensource.org/history>

- Definice open source (1998):
  - <https://opensource.org/osd-annotated>
  - Kritéria pro určení, jestli je software (přesněji jeho **licence**) open source nebo ne.
  - Vznikla zobecněním „Debian Free Software Guidelines“ (1997)
- Co na to RMS?

Pojem „open source“ software je používán některými lidmi pro pojmenování víceméně stejně kategorie software jako je svobodný software. Není to ale přesně to samé: oni **tolerují některé licence**, které my požadujeme za příliš omezující a existují licence svobodného software, které oni neakceptují. Rozdíly jsou ale malé: **téměř všechn svobodný software je open source a naopak.**

<https://www.gnu.org/philosophy/categories.html>

# Duševní vlastnictví

- WIPO: World Intellectual Property Organization – agentura v rámci OSN
- **Duševní vlastnictví** jsou **výsledky procesu lidské tvořivosti, zkoumání a myšlení** jako například vynálezy, literární a umělecká díla, design, a symboly, jména a obrázky používané v obchodu.

<https://www.wipo.int/about-ip/en/>

- Software spadá pod duševní vlastnictví
- Lze jej chránit pomocí:
  - patentového práva,
  - ochranných známek,
  - zákonů o obchodním tajemství a
  - autorského práva (copyright).

# Obchodní tajemství

[https://www.wipo.int/sme/en/ip\\_business/trade\\_secrets/trade\\_secrets.HTML](https://www.wipo.int/sme/en/ip_business/trade_secrets/trade_secrets.HTML)

- Nejstarší forma duševního vlastnictví
- Jedná se o informaci, která:
  - má obchodní hodnotu vyjádřitelnou penězi,
  - není v příslušných obchodních kruzích běžně známa,
  - je majitelem aktivně utajována.
- **Příklad:** recept na Coca-Colu; proprietární software
- Ochrana pomocí obchodního tajemství **se získá prohlášením**, že detaily daného předmětu jsou tajné.
- Obchodní tajemství **trvá** po dobu jeho aktivní ochrany. **Lze ho znehodnotit** prozrazením, zpětným inženýrstvím či nezávislým objevem.
- Pro FOSS nepoužíváno :-)

# Ochranné známky

- Ochranné známky chrání vztah mezi *slovem, frází, symbolem či designem* a poskytovatelem zboží či služeb.
- Jsou určeny k **ochraně spotřebitele** před zmatením ohledně toho, od koho kupují produkt či službu.
  - Použití známky k pravdivému označení produktu/služby je zákonem dovoleno (tzv. *nominative [fair] use*)
- Ochranné známku lze získat *de facto* pouhým používáním, její **registrace** (v ČR na úřadu průmyslového vlastnictví) dává majiteli další práva.
- Ochranné známky mohou trvat věčně, pokud jsou používány a silné na trhu
  - Např. Jak dlouho budou existovat známky jako McDonald's, Coca-Cola, Apple nebo ČVUT?
- Ochrana pomocí ochranných známek je omezena na segment trhu (např. jídlo, počítače) a území (Evropa, USA, ...).

# Open source a ochranné známky

Open Source Definition §4:

## Integrita autorova zdrojového kódu

*Licence smí omezovat distribuci modifikovaného zdrojového kódu pouze pokud dovoluje distribuci „patchů“ společně se zdrojovým kódem, za účelem modifikace programu při komplikaci. Licence musí explicitně dovolovat distribuci software sestaveného z modifikovaného zdrojového kódu. **Licence může požadovat, aby odvozené dílo neslo jiné jméno nebo číslo verze než originál.***

<https://opensource.org/osd>

- ⇒ Ochranné známky jsou s open source kompatibilní
- Příklad: Firefox – ochranné známky zabránily šíření modifikované verze s malwarem.

# Patenty

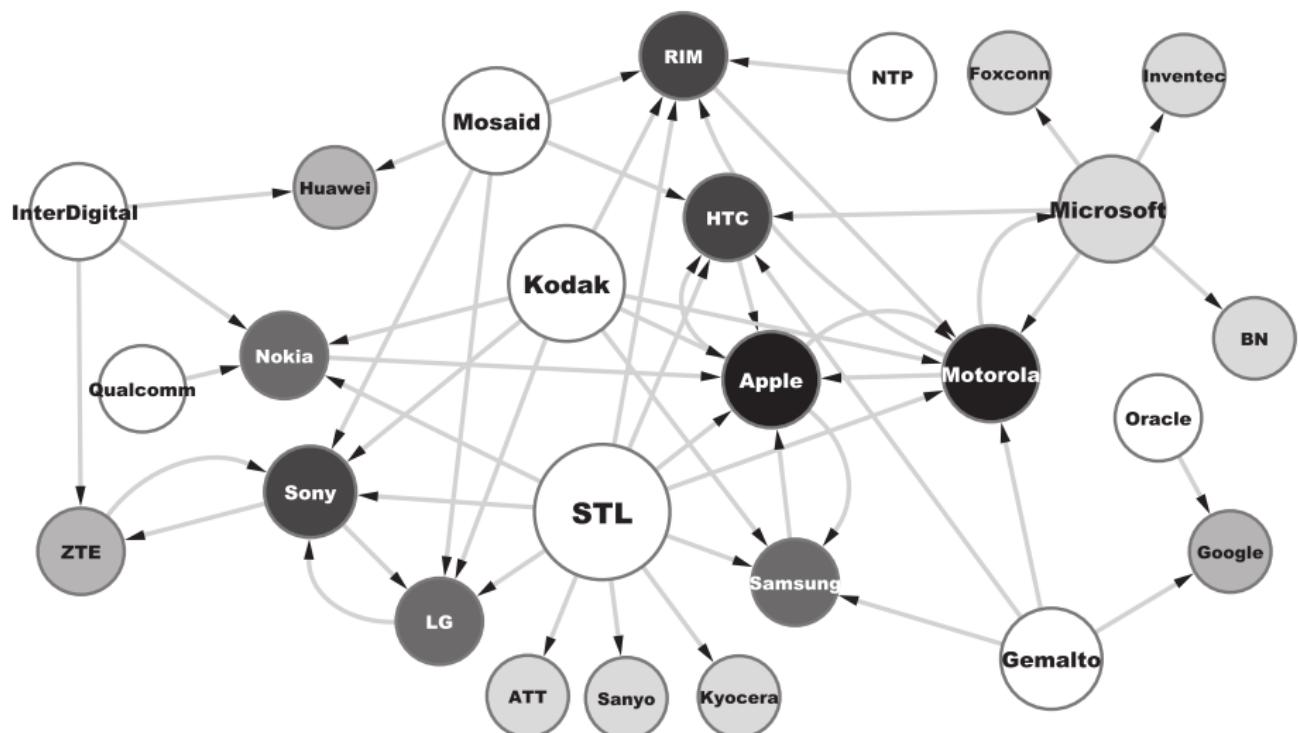
- Patent zaručuje jeho majiteli ochranu vlastnických práv k **vynálezu či technickému řešení** na zařízeních či procesech poskytující „užitečnou“ funkci.
- Patent poskytuje vynálezci **časově omezený monopol** (např. 20 let) na výrobu, použití, prodej či dovoz vynálezu.
- Monopol se vztahuje na ostatní i v případě nezávislého objevu.
- Výměnou za patentovou ochranu musí vynálezce plně zveřejnit svůj vynález v patentové přihlášce (která je dostupná všem).
- Ostatní musí pro využití vynálezu získat od vynálezců licenci (většinou za peníze).

## Patenty a FOSS

- Patenty představují hrozbu pro uživatele FOSS (a nejen pro ně).
- Některé licence proti tomu poskytují ochranu (viz dále)

# Mobilní patentové války

Patent-infringement lawsuits in the mobile phone industry in 1/2006 – 8/2011



# Autorské právo

- Omezuje použití specifických **vyjádření** myšlenek
- t.j., může být použito k omezení činností jako např.:
  - Tvorba **kopií** díla a jejich prodej
  - Tvorba **odvozených** děl
  - **Veřejné provozování** díla
  - **Prodej** a převod autorských práv na jiné osoby
- Vztahuje se na cokoli, co vykazuje známky kreativity
- **Automaticky se vztahuje na všechno co vytvoříte**, jakmile je to zafixováno v nějaké konkrétní podobě.
  - Výchozí nastavení ≈ autor má monopol, ostatní nemohou s dílem nakládat
- Omezení:
  - Trvání: obvykle 70–150 let (podle státu)
  - Férové použití: svoboda slova, svoboda citování atd. (v českých zákonech toto omezení není)
- Téměř shodné po celém světě – od Bernské úmluvy z roku 1886

# Proč jsou potřeba licence?

- Autorské právo platí pro software
- Výchozí nastavení je „**všechna práva vyhrazena**“

## Z pohledu uživatele

- Bez licence nemůžete se softwarem dělat skoro nic.

## Z pohledu autora

- Bez licence nemohou vaší (potenciální) uživateli SW používat
- Musíte jim dát aspoň nějaká práva

# Licence svobodného software a autorské právo

- Licence svobodného software jsou **právní „hack“** (nazývaný copyleft):
  - jsou podobné ostatním licencím, ale místo toho, aby činnost uživatelů omezovaly, tak naopak uživatelům některá „speciální“ práva dávají
- Licence FOSS dávají uživatelům právě ta práva, aby si uživatelé **mohli užívat 4 základní svobody** (spouštět, studovat, kopírovat a modifikovat)
- Ale to neznamená, že je s FOSS možné dělat cokoli
  - FOSS licence většinou vyžadují plnění **určitých podmínek**
  - Pokud je uživatel nedodrží, licence pro něj neplatí a tudíž platí výchozí nastavení autorského práva „všechna práva vyhrazena“.

## Poznámka

FOSS **není proti ochraně duševního vlastnictví**. Ve skutečnosti licence FOSS využívají autorského práva k zajištění svobody SW.

# Copyleft

Omezující pravidlo, které zajistí, že při šíření programu nelze přidat omezení, která odepřou jiným lidem 4 základní svobody: **spouštět, studovat, kopírovat a modifikovat**. Toto omezení není v konfliktu se svobodou; naopak svobodu chrání.

- **Copyleft využívá autorského zákona**, který umožňuje autorům definovat omezení pro užití jejich díla.
- Konkrétně bývá copyleft implementován pomocí různých klauzulí v licencích, jako **GNU General Public License (GPL)** nebo **Creative Commons Attribution Share Alike License** (viz dále).
- Copyleft klauzule zajišťuje, že každý uživatel daného programu, bez ohledu na to zda/jak je někým modifikován, se může těšit ze 4 svobod. **Při šíření programu musí každý tyto svobody zachovat.**

# Kategorie FOSS licencí

Licence FOSS mohou být **klasifikovány podle podmínek, které vyžadují** výměnu za svobodu softwaru. Obecně se hovoří o následujících třídách licencí FOSS:

- **Permisivní:** BSD, MIT, Apache, ...
- **Reciproční:** GPL, ... („silný copyleft“)
- **Reciproční s omezeným působením:** LGPL, MPL, ... („slabý copyleft“)

## Poznámky

- Striktnější licence (např. GPL) nejsou „méně svobodné“ než jiné (např. BSD)
- I nejstriktnější FOSS licence dovolují nepoměrně víc než licence proprietárního software (např. EULA (End User License Agreement) z Microsoft Windows)

## Akademické licence

- Významná podmnožina populárních permisivních licencí
- Původně vytvořeny a popularizovány univerzitami
- Jedná se o nejjednodušší licence: velmi málo omezení
- Vyžadují uvedení původu (**ponechání jmen autorů a hlášek „copyright“**)
- Lze použít k čemukoli, včetně **použití v proprietárním produktu**

### Příklady:

- MIT – jQuery
- BSD – FreeBSD, CMake, OpenCV, Redis
- ISC – BIND, ISC DHCP

# Permisivní licence

- Nadmnožina akademických licencí
- Explicitně udělují patentové licence (v moderních variantách)
  - Nemůže se stát, že autor tajně vloží do FOSS kus chráněný patentem, a když začnete software využívat (a vydělávat na něm), tak vás zažaluje za porušení patentu.
- Lze použít k čemukoliv, včetně **použití v proprietárním produktu**

## Příklady:

- Apache License – Apache web server, Ant, TensorFlow

# Reciproční licence

- Požadují, aby **odvozená díla měla stejnou licenci**
- V mnoha případech požadují, aby při **šíření software** v binární podobě byl zahrnut i **kompletní zdrojový kód**
- Také známo jako „silný copyleft“ nebo jen „**copyleft**“
- Občas **hanlivě označované jako „virální licence“**
  - Pokud je kód licencovaný reciproční licencí zahrnut do nějaké aplikace, pak je aplikace „infikována“ a musí být **celá** uvolněna pod reciproční licencí.

## Příklady:

- GNU GPL – Linux a mnoho dalších známých projektů
- AGPL – MongoDB, CiviCRM
- CC BY-SA (pro nesoftwareová díla) – tato prezentace

## Reciproční licence s omezeným působením

- Jako reciproční licence, ale s **omezeným rozsahem působnosti** – jaké části odvozeného díla spadají pod původní licenci
  - Změny **hlavního díla** spadají pod stejnou licenci
  - **Ostatní díla**, která jsou použita s/přidána k/zahrnují hlavní(mu) dílo(u) nemusí mít stejnou licenci
- Různé licence se liší způsobem, jakým je rozsahem hlavního díla omezen
- Dle hanlivé analogie: „viralita“ licence je omezena na hlavní dílo
- Také známo jako: „slabý copyleft“

### Příklady:

- LGPL – působnost omezena na knihovnu – Qt
- MPL – působnost omezena na zdrojový soubor – Firefox, Libre Office
- CDDL – NetBeans

# Omezení v licencích FOSS

Je možné, aby byly v licencích FOSS nějaká omezení?

Ano! Vše, co nebrání svobodě softwaru je možné omezit.

Typická omezení ve FOSS licencích:

- **Uvedení autora** – není možné z díla odstranit zmínku o originálním autorovi
- **Přenos svobody** – t.j. copyleft
- Detaily ochrany **uživatelských svobod** – přístup ke zdrojovému kódu, zákaz „technických opatření“ typu DRM, ...

# Populární a významnější licence

## Doporučení

- **Nevytvářejte svou vlastní licenci**, uděláte to špatně!
- Zvolte si strategii a pak si vyberte z existující nabídky licencí.

### ■ Permisivní

- BSD 3-Clause "New" or "Revised" license
- BSD 2-Clause "Simplified" or "FreeBSD" license
- Apache License 2.0
- MIT license
- ISC License

### ■ Reciproční (AKA "strong copyleft")

- GNU General Public License (GPL), versions 2 and 3
- GNU Affero General Public License (AGPL), version 3

### ■ Reciproční s omezeným působením (AKA "weak copyleft")

- GNU Lesser General Public License (LGPL), versions 2.1 and 3
- Mozilla Public License (MPL), version 2.0
- Common Development and Distribution License (CDDL)

# http://choosealicense.com/ by GitHub

License	Commercial Use	Distribution	Modification	Patent Use	Private Use	Disclose Source	License and Copyright Notice	Network Use is Distribution	Same License	State Changes	Hold Liable	Trademark Use
Academic Free License v3.0	●	●	●	●	●		●			●	●	●
GNU Affero General Public License v3.0	●	●	●	●	●	●	●	●	●	●	●	●
Apache License 2.0	●	●	●	●	●		●			●	●	●
Artistic License 2.0	●	●	●	●	●		●			●	●	●
BSD 2-clause "Simplified" License	●	●	●		●		●				●	
BSD 3-clause Clear License	●	●	●	●	●		●				●	
BSD 3-clause "New" or "Revised" License	●	●	●		●		●				●	
Creative Commons Attribution 4.0	●	●	●	●	●		●			●	●	●
Creative Commons Attribution Share Alike 4.0	●	●	●	●	●		●		●	●	●	●
Creative Commons Zero v1.0 Universal	●	●	●	●	●					●	●	●
Eclipse Public License 1.0	●	●	●	●	●	●	●		●			●
European Union Public License 1.1	●	●	●	●	●	●	●	●	●	●	●	●
GNU General Public License v2.0	●	●	●		●	●	●		●	●	●	●
GNU General Public License v3.0	●	●	●	●	●	●	●		●	●	●	●
ISC License	●	●	●		●		●				●	
GNU Lesser General Public License v2.1	●	●	●		●	●	●		●	●	●	●
GNU Lesser General Public License v3.0	●	●	●	●	●	●	●		●	●	●	●
LaTeX Project Public License v1.3c	●	●	●		●	●	●			●	●	●
MIT License	●	●	●		●		●				●	
Mozilla Public License 2.0	●	●	●	●	●	●	●			●	●	●

<https://tldrlegal.com/>

## Browse Software Licenses & Summaries

Lookup License by Conditions

### FEATURED

[YouTube Terms of Service](#) 1476 ⓘ 1 ❤️  
Terms of Service managed by [seldon](#)

[Sleepycat License](#) 2184 ⓘ 2 ❤️  
Code License managed by [valeriedouglas](#)  
[3 Rules](#) [2 Rules](#) [2 Rules](#)

[Fair Source License 0.9 \(Fair-Source-0.9\)](#) 1685 ⓘ 3 ❤️  
Code License managed by [kevin](#)  
[6 Rules](#) [3 Rules](#) [3 Rules](#)

[Mozilla Public License 1.0 \(MPL-1.0\)](#) 3920 ⓘ 2 ❤️  
Code License managed by [bfitzg](#)  
[5 Rules](#) [1 Rules](#) [4 Rules](#)

[GNU Lesser General Public License v3 \(LGPL-3.0\)](#) 30258 ⓘ 5 ❤️  
Code License managed by [kevin](#)  
[5 Rules](#) [2 Rules](#) [6 Rules](#)

[Minecraft End User Licence Agreement](#) 13707 ⓘ 6 ❤️  
Terms of Service managed by [ItsBobberson](#)

### MOST POPULAR

[MIT License \(Expat\)](#) 196025 ⓘ 64 ❤️  
Code License managed by [kevin](#)  
[5 Rules](#) [1 Rules](#) [2 Rules](#)

[Apache License 2.0 \(Apache-2.0\)](#) 193363 ⓘ 52 ❤️  
Code License managed by [kevin](#)  
[7 Rules](#) [2 Rules](#) [4 Rules](#)

[GNU General Public License v3 \(GPL-3\)](#) 125230 ⓘ 36 ❤️  
Code License managed by [kevin](#)  
[5 Rules](#) [2 Rules](#) [6 Rules](#)

[GNU General Public License v2.0 \(GPL-2.0\)](#) 53893 ⓘ 10 ❤️  
Code License managed by [kevin](#)  
[4 Rules](#) [2 Rules](#) [5 Rules](#)

[GNU Lesser General Public License v2.1 \(LGPL-2.1\)](#) 49172 ⓘ 8 ❤️  
Code License managed by [kevin](#)  
[3 Rules](#) [2 Rules](#) [6 Rules](#)

[BSD 2-Clause License \(FreeBSD/Simplified\)](#) 48540 ⓘ 9 ❤️  
Code License managed by [kevin](#)  
[4 Rules](#) [1 Rules](#) [2 Rules](#)

### NEWEST

[PHP License 3.0.1](#) 11 ⓘ 1 ❤️  
Code License managed by [wilfred](#)

[Muneto - License](#) 11 ⓘ 0 ❤️  
Code License managed by [AngelKrak](#)

[Freeware](#) 20 ⓘ 0 ❤️  
Code License managed by [surkum](#)

[TRDRLegal terms of servis](#) 33 ⓘ 1 ❤️  
Code License managed by [widarto](#)

[TestCocopods](#) 45 ⓘ 0 ❤️  
Code License managed by [Yuan](#)

[Google APIs Terms of Service](#) 100 ⓘ 0 ❤️  
Code License managed by [jmeng](#)

## 3-clause BSD (1999)

Copyright (c) <year>, <copyright holder>  
All rights reserved.

Redistribution and use in source and binary forms, with or without  
modification, are permitted provided that the following conditions are met:

- \* Redistributions of **source code must retain the above copyright notice, this list of conditions and the following disclaimer.**
- \* Redistributions in **binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.**
- \* Neither the name of the <organization> nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

[...]

- popular permissive license (used e.g. by CMake, ...)
- you may redistribute the work, in any form (source or binary), with or without modifications, as long as you **preserve copyright notices**
- non endorsement requirement
- GPL compatible

# The MIT License (1988)

*Copyright (c) <year> <copyright holders>*

*Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to **use, copy, modify, merge, publish, distribute, sublicense, and/or sell** copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:*

*The above **copyright notice** and this permission notice shall be included in all copies or substantial portions of the Software.*

[...]

- functionally similar to modern BSD licenses
- used by X11/X.org, Symfony, RoR, Lua, Putty, Mono, jQuery, ...
- no explicit non endorsement clause
- it states more explicitly the rights given to the end-user
- GPL compatible

# GNU General Public License (GPL)

1989 verze 1 (by RMS), zobecnění (generalization – odtud pochází její jméno) licencí použitými projekty GNU: Emacs, GDB, GCC

1991 verze 2 (by RMS)

- „svoboda nebo smrt“; raný příklad obrany proti softwarovým patentům a jiným omezením uživatelských svobod

2007 verze 3 (by RMS s pomocí E. Moglen/SFLC)

- Veřejný proces revizí
- Klauzule o softwarových patentech
- Klauzule o DRM (anti „tivoization“)
- Snaha o kompatibilitu s podobnými licencemi
- Internacionálizace (jiné právní systémy než USA)
- Ochrana licence proti dalším omezením
- ⇒ verze 2 je mnohem jednodušší

## GPLv2 – copyleft

<https://www.gnu.org/licenses/old-licenses/gpl-2.0.cs.html>

2. Můžete **upravovat** vaši kopii, či kopie Programu, anebo kterékoliv jeho části a tak vytvořit dílo založené na Programu, a kopírovat a šířit takové úpravy, či dílo, podle podmínek odstavce 1 výše, za předpokladu, že splníte všechny tyto podmínky:
  - a) Upravené soubory **musíte opatřit zřetelnou zmínkou** uvádějící, že jste soubory změnily s datem každé změny.
  - b) Musíte umožnit, aby jakékoli vámi **zveřejněné či šířené dílo**, které jako celek nebo zčásti obsahuje Program nebo jakoukoli jeho část, popřípadě je z Programu nebo jeho části odvozeno, mohlo být jako celek bezplatně poskytnuto každé třetí osobě v **souladu s ustanoveními této Licence**.

# GPLv2

## Požadavek na zdrojový kód

3. Můžete kopírovat a šířit Program (nebo dílo na něm založené, viz odstavec 2) pomocí strojového kódu anebo ve spustitelné podobě podle ustanovení odstavců 1 a 2 výše, pokud splníte některou z následujících náležitostí:
  - a) Doprovoďte jej **strojově čitelným zdrojovým kódem**, který musí být šířen podle ustanovení odstavců 1 a 2 výše, a to na médiu běžně používaném pro výměnu softwaru, nebo
  - b) ...
  - c) ...

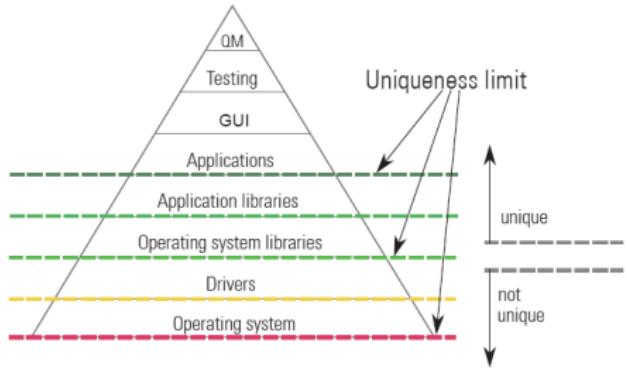
Zdrojový kód k dílu je **nejvhodnější formou díla pro jeho případné úpravy**. Pro spustitelné dílo úplný zdrojový kód znamená **veškerý zdrojový kód pro všechny moduly**, které obsahuje, včetně jakýchkoli dalších souborů pro definici rozhraní a **dávkových souborů potřebných pro komplikaci a instalaci** spustitelného díla. Zvláštní **výjimkou** jsou části, které **jsou běžně šířeny** (ve zdrojové nebo binární formě) s hlavními součástmi (kompilátor, jádro, apod.) operačního systému, na němž spustitelné dílo běží, pokud ovšem taková část sama nedoprovází spustitelné dílo.

## GPL stručně

- Software pod GPL můžete i prodávat (RedHat)
- Musíte zajistit, že se SW vždy dodáte i zdrojové kódy **kompletního** programu, aby si příjemce programu mohl SW upravit
- Pokud používáte nějaké knihovny, musí být kompatibilní s GPL (jejich licence není v rozporu s GPL, např. musíte mít právo je distribuovat včetně zdrojových kódů)
- Pokud program nešíříte, můžete si s ním dělat co chcete

# Otevřít nebo neotevřít?

- Firemní know-how:
  - konkurenční výhoda (malá část)
  - ostatní (velká část)
- Maximalizace ekonomického úspěchu = maximalizace investic do konkurenční výhody
- Získání ostatního know-how:
  - koupit ho
  - spolupracovat s ostatními (i s konkurenty) → **FOSS**
    - Vede ke standardizaci = nižším nákladům
- Globalizace
  - Generuje tlak na firmy – více konkurence, levnější práce jinde
  - Zároveň umožňuje vývoj FOSS
- FOSS lze chápat jako kompenzaci negativních efektů globalizace



Zdroj: OSADL

# Jak vydělat peníze s FOSS?

Mnoho způsobů – nejběžnější:

- 1 Prodej HW:** Dostupnost FOSS zvyšuje užitečnost/prodej HW
  - Android, Intel, ...
- 2 Placená podpora:** SW je zdarma; lidé platí za vývojářovo know-how (konzultace) nebo vývoj rozšíření SW (také FOSS)
  - Cygnus = *Cygnus, your GNU Support* (firma stojící za překladačem GCC, překladač pro PlayStation), RedHat
- 3 Dvojité licencování:** copyleftová licence nebo placená komerční licence
  - Pokud firma nechce nebo nemůže použít copyleft licenci, zaplatí
  - Qt, MySQL
- 4 Reklama:** FOSS dělá reklamu jinému produktu/firmě a je jím placen
  - Firefox, (Ubuntu)
- 5 Placená rozšíření:**
  - Eclipse IDE

# Komerční versus open source řešení

## Komerční řešení

- Vendor lock-in
- Při krachu firmy máte většinou smůlu
- Operační systémy: Stále více lidí používá FOSS ⇒ (menší) komerční OS (VxWorks, QNX, ...) jsou dražší

## FOSS řešení

- Potřebujete zkušenosti, znalosti
- Je dobré rozumět „FOSS kultuře“ (viz další slide)
- Možnost najmout si firmu, která vám FOSS nasadí

# Zapojení se do FOSS projektu

- Přihlaste se do fóra, mailing listu, IRC; sledujte „watch“ repo na GitHubu , ...
- Čtěte komunikaci vývojářů (fórum, bug tracker, ...), zjistěte, jak projekt funguje.
  - Mnoho projektů zveřejňuje dokument „Jak přispět“
- Představte se, ptejte se
  - “Asking smart questions” FAQ (Eric S. Raymond)  
<http://www.catb.org/~esr/faqs/smart-questions.html>
- Neplýtvejte časem správců, reagujte rychle
- Občas je lepší poslat rovnou „RFC<sup>1</sup> patch/pull request“, než se ptát
  - Ukazujete, že jste ochotni investovat do řešení problémů
  - Modifikováním SW mnoho věcí zjistíte sami
  - Stanete se lepším partnerem pro diskuzi
- Pokročilá účast na projektu
  - Posílejte vaše patche/pull requesty
  - Komentujte patche/PR ostatních
  - Překládejte, dokumentujte, kreslete, ...

<sup>1</sup>Request for comments

# Závěr

- FOSS se ukázalo jako výborný způsob vývoje software
- Už i Microsoft používá, propaguje a vyvíjí open source
- Mladá generace (vy) už nevnímá užitečnost copyleftu tak výrazně, protože open source je dnes normální ⇒ vzestup permisivních licencí (MIT, Apache).
  - Byl by IT svět tam, kde teď je, kdyby neexistovala licence GPL? (Viz problémy BSD UNIXu v minulosti...)