

02. Koncepce OS

Procesy, vlákna

ZOS 2024

L. PEŠIČKA

Scénář příkladu (!)

1. Uživatel Pepa spustí program (textový editor), který poběží jako proces **p1**.
2. Proces bude chtít otevřít soubor **ahoj.txt**.
3. O otevření souboru musí proces požádat operační systém **systémovým voláním open()**.
4. Soubor **ahoj.txt** bude ve filesystemu chráněný pomocí **ACL (Access Control List)**, kdo k němu smí přistoupit.
5. Jádro operačního systému zkontroluje, zda jej smí otevřít, a pokud ano, soubor otevře (naplní příslušné datové struktury).

uživatel Pepa (UID= 1015)

CPU ví, zda je v uživatelském nebo
privilegovaném režimu

proces p1 (PID = 202)
volá systémové volání
open("ahoj.txt")

Tabulka vektorů
přerušení

uživatelský
režim

odpovídající instrukce procesu:
EAX <- číslo služby open
EBX, ECX, .. <- další param.
INT 0x80

index 128

adresa

privilegovaný
režim

vstupní bod jádra, dle EAX
zavolá příslušné volání

1023

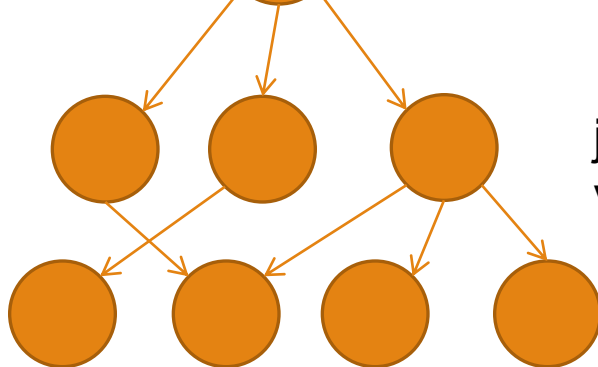
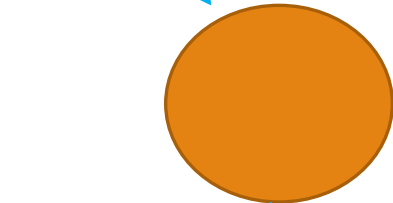
RAM

256 indexů
0 .. 255

každá položka
obsahuje
adresu
obslužné rutiny

jednotlivá systémová
volání (rozhodne EAX)

Tabulka vektorů přerušení je v RAM
kde leží udává registr IDTR



Jak jádro rozhodne, že má uživatel k souboru přístup?

Implementace volání `open()` zjistí:

- na kterém filesystemu (fs) soubor ahoj.txt leží
`ntfs, fat32, ext3, ext4, xfs, ...`
- zda daný fs podporuje ACL (komplexní práva) nebo základní unixová práva (vlastník, skupina, ostatní) nebo žádná kontrola práv (FAT)
`ACL slouží ke kontrole přístupových práv`
- zkontroluje, zda ACL vyhovují pro daného uživatele a daný mód otevření souboru (uid, čtení/zápis)

ACL

Kontrolou přístupových práv jádro zjistí, že uživatel Pepa má právo daný soubor otevřít a požadovanou akci vykoná

ahoj.txt



ACL tabulka práv příslušející k souboru ahoj.txt

Uživatel (0)/ skupina (1)	identifikátor	práva
0	1015 (Pepa)	R, W
0	1018 (Tomáš)	R
1	104 (studenti)	R
1	105 (externisté)	R

Obsluha přerušení (!!!)

I. Mechanismus vyvolání přerušení (vyvolání instrukcí: **INT číslo**)

- Na zásobník se uloží registr příznaků **FLAGS**
- Zakáže se přerušení (vynuluje příznak IF – Interrupt Flag v registru FLAGS)
- Na zásobník se uloží návratová adresa (obsah registrů **CS:IP**, obecně PC – program counter) ukazující na instrukci, kde budeme po návratu z přerušení pokračovat

II. Kód obsluhy přerušení – „píše programátor OS“

- Na zásobník uložíme hodnoty registrů (abychom je procesu nezměnili)
- Vlastní kód obsluhy (musí být rychlý, případně naplánujeme další věci)
- Ze zásobníku vybereme hodnoty uschovaných registrů (aby přerušený proces nic nepoznal)
- (Někdy ale např. u sw přerušení chceme, aby v nám v nějakém registru předal OS výsledek)

III. Návrat z přerušení (instrukce: **IRET**)

- Ze zásobníku je vybrána návratová adresa (obsah registrů **CS:IP**) – kde budeme pokračovat
- Ze zásobníku se obnoví registr **FLAGS** – obnoví původní stav povolení přerušení

Poznámka

– tabulka vektorů přerušení

Přerušení a výjimky vznikají a obsluhují se v reálném režimu téměř shodně s procesorem 8086. Rozlišuje se 256 různých přerušení a výjimek. Pro každé přerušení nebo výjimku je v paměti uloženo 32 bitů adresy (přerušovací vektor) začátku obslužné programové rutiny. Adresy jsou zapsány v **tabulce přerušovacích vektorů** (viz obr. 4.2). Tabulka má velikost 1 KB a je implicitně uložena na začátku paměti od adresy 0000:0000.

31	0	Adresa	Číslo přer. vektoru
<i>segment</i>	<i>offset</i>	0:03FC	INT 0FFh
	⋮	⋮	
<i>segment</i>	<i>offset</i>	0:000C	INT 3
<i>segment</i>	<i>offset</i>	0:0008	INT 2
<i>segment</i>	<i>offset</i>	0:0004	INT 1
<i>segment</i>	<i>offset</i>	0:0000	INT 0

Zdroj:
M. Brandejs
Mikroprocesory
Intel Pentium

Popis zde
odpovídá
pro OS MS DOS

Obr. 4.2 Tabulka přerušovacích vektorů reálného režimu

Poznámka - podrobněji

Po přijetí žádosti o přerušení provádí procesor v reálném režimu tyto akce:

1. do zásobníku se uloží registr příznaků (FLAGS),
2. vynulují se příznaky IF a TF,
3. do zásobníku se uloží registr CS,
4. registr CS se naplní 16bitovým obsahem adresy $n \times 4 + 2$,
5. do zásobníku se uloží registr IP ukazující na neprovedenou instrukci,
6. registr IP se naplní 16bitovým obsahem adresy $n \times 4$.

Výjimky v reálném režimu nevracejí chybový kód. Návrat do přerušného procesu a jeho pokračování zajistí instrukce IRET, která provede činnosti v tomto pořadí:

1. ze zásobníku obnoví registr IP,
2. ze zásobníku obnoví registr CS,
3. ze zásobníku obnoví příznakový registr (FLAGS).

Zdroj:
M. Brandejs
Mikroprocesory
Intel Pentium

Poznámka - IF

IF (Interrupt Enable Flag) vynulovaný instrukcí CLI zabrání uplatnění vnějších maskovatelných přerušení (generovaných signálem INTR). Nastavení příznaku na jedničku (instrukcí STI) přerušení povoluje. Maskovat lze přerušení od vnějších zařízení (klávesnice, tiskárna atd.), nikoli výjimky, programová přerušení (INT) a NMI (přerušení ze skupiny nemaskovatelných přerušení). Hodnoty CPL a IOPL určují, zda lze tento příznak měnit instrukcemi CLI, STI, POPF, POPFD a IRET.

Zdroj:
M. Brandejs
Mikroprocesory
Intel Pentium

Poznámka - IF

Interrupt flag

From Wikipedia, the free encyclopedia

IF (Interrupt Flag) is a system flag bit in the x86 architecture's **FLAGS register**, which determines whether or not the **CPU will handle maskable hardware interrupts**.^[1]

The bit, which is bit 9 of the **FLAGS register**, may be set or cleared by programs with sufficient privileges, as usually determined by the Operating System. If the flag is set to 1, maskable hardware interrupts will be handled. If cleared (set to 0), such interrupts will be ignored. IF does not affect the handling of non-maskable interrupts or software interrupts generated by the INT instruction.

Contents [\[hide\]](#)

- 1 Setting and clearing
- 2 Privilege level
 - 2.1 Old DOS programs
- 3 CLI
- 4 STI
- 5 See also
- 6 References
- 7 External links

Setting and clearing [\[edit \]](#)

The flag may be set or cleared using the CLI (Clear Interrupts), STI (Set Interrupts) and POPF (Pop Flags) instructions.

CLI clears IF (sets to 0), while STI sets IF to 1. POPF pops 16 bits off the stack into the **FLAGS register**, which means IF will be set or cleared based on the ninth bit on the top of the stack.^[1]

Registr FLAGS
procesoru

Každý bit má svůj
vlastní význam.

9. bit je IF

Intel x86 FLAGS register ^[1]			
Bit #	Abbreviation	Description	Category
FLAGS			
0	CF	Carry flag	Status
1		Reserved	
2	PF	Parity flag	Status
3		Reserved	
4	AF	Adjust flag	Status
5		Reserved	
6	ZF	Zero flag	Status
7	SF	Sign flag	Status
8	TF	Trap flag (single step)	Control
9	IF	Interrupt enable flag	Control
10	DF	Direction flag	Control
11	OF	Overflow flag	Status
12-13	IOPL	I/O privilege level (286+ only), always 1 on 8086 and 186	System
14	NT	Nested task flag (286+ only), always 1 on 8086 and 186	System
15		Reserved, always 1 on 8086 and 186, always 0 on later models	
EFLAGS			
16	RF	Resume flag (386+ only)	System
17	VM	Virtual 8086 mode flag (386+ only)	System
18	AC	Alignment check (486SX+ only)	System
19	VIF	Virtual interrupt flag (Pentium+)	System
20	VIP	Virtual interrupt pending (Pentium+)	System
21	ID	Able to use CUID instruction (Pentium+)	System

Úvaha – zákaz přerušení

Proč se při obsluze přerušení zakazuje další přerušení?

- Představte si např., že ošetřujete **HW přerušení vstup z klávesnice**.
 - Stisknete jednu klávesu a pustí se obsluha přerušení.
 - Pokud by nedošlo k zakázání přerušení, tak než by obsluha přerušení doběhla a byla by stisknuta další klávesa, k čemu by mohlo dojít?
 - Původní „rozpracované“ přerušení by bylo přerušeno dalším a to obvykle nechceme. Zpracujeme danou obsluhu a až poté se opět povolí další přerušení (IF flag se nastaví na původní hodnotu).

Komentáře - maskování

- SW přerušení
 - vyvoláme v programu instrukcí INT
 - Nemůžeme jej zamaskovat
- HW přerušení
 - od nějakého zařízení (časovač, klávesnice)
 - Lze jej zamaskovat (tj. zařízení požádá ale v danou chvíli ignorujeme)
- Vnitřní přerušení
 - výjimky – dělení nulou, neplatná instrukce, výpadek stránky paměti
 - Nemůžeme je zamaskovat (tvářit se, že výjimka není?)

Příklad mapování vektorů přerušení

Tabulka
vektorů
přerušení

0

Adresa
obsluhy

1

Adresa
obsluhy

2

Adresa
obsluhy

254

Adresa
obsluhy

255

Adresa
obsluhy

Kód	Číslo přerušení	Popis
B	INT 00H	Dělení nulou
B	INT 01H	Krokování
B	INT 02H	Nemaskovatelné přerušení
B	INT 03H	Bod přerušení (breakpoint)
B	INT 04H	Přetečení
B	INT 05H	Tisk obrazovky
B	INT 06H	Nesprávný operační kód
B	INT 07H	Není koprocessor
B	INT 08H IRQ0	Přerušení od časovače
B	INT 09H IRQ1	Přerušení od klávesnice
	INT 0aH IRQ2	EGA vertikální zpětný běh
	INT 0bH IRQ3	COM2
	INT 0cH IRQ4	COM1
	INT 0dH IRQ5	Přerušení harddisku
B	INT 0eH IRQ6	Přerušení řadiče disket
	INT 0fH IRQ7	Přerušení tiskárny
B	INT 10H	Služby obrazovky
	INT 11H	Seznam vybavení
	INT 12H	Velikost volné paměti
B	INT 13H	Diskové vstupně-výstupní operace

Příklad možného
mapování
(původní IBM PC) ,
může být různé

dva pojmy:

INT ... „index“

IRQ ... „drát“

všimněte si IRQ0 je
zde na INT 08H, na
vektoru 08H (tj.od
adresy 8*4) bude
adresa podprogramu
k vykonání dané
obsluhy

IRQ – Interrupt Request

IRQ – signál, kterým zařízení (**časovač**, **klávesnice**) žádá procesor o přerušení zpracovávaného procesu za účelem provedení obsluhy požadavku zařízení

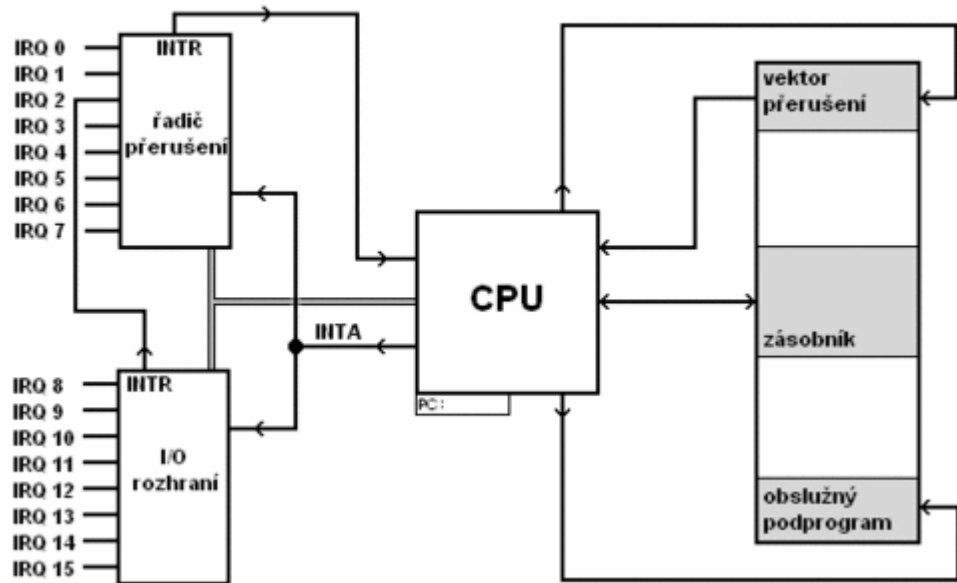
IRQL – priorita přerušovacího požadavku
(Interrupt Request Level)

NMI – nemaskovatelné přerušení, např. nezotavitelná hw chyba
(non-maskable interrupt)

Obsluha HW přerušení

1. zařízení sdělí **řadiči přerušení**, že potřebuje přerušení
2. řadič upozorní CPU, že jsou čekající (**pending**) přerušení
3. až je CPU ochotné přijmout přerušení (dokončí rozpracovanou instrukci) tak přeruší výpočet a zeptá se řadiče přerušení, které nejdůležitější čeká a od řadiče dostane po datové sběrnici index do tabulky vektorů přerušení
4. CPU spustí obsluhu přerušení
5. uloží stav procesu (automaticky návratová adresa a FLAGS, ostatní si musí ošetřit), provede základní obsluhu zařízení, informuje řadič o dokončení obsluhy, obnoví stav procesu a pokračuje se dále

Řadič přerušení



2 integrované obvody Intel 8259 v kaskádě

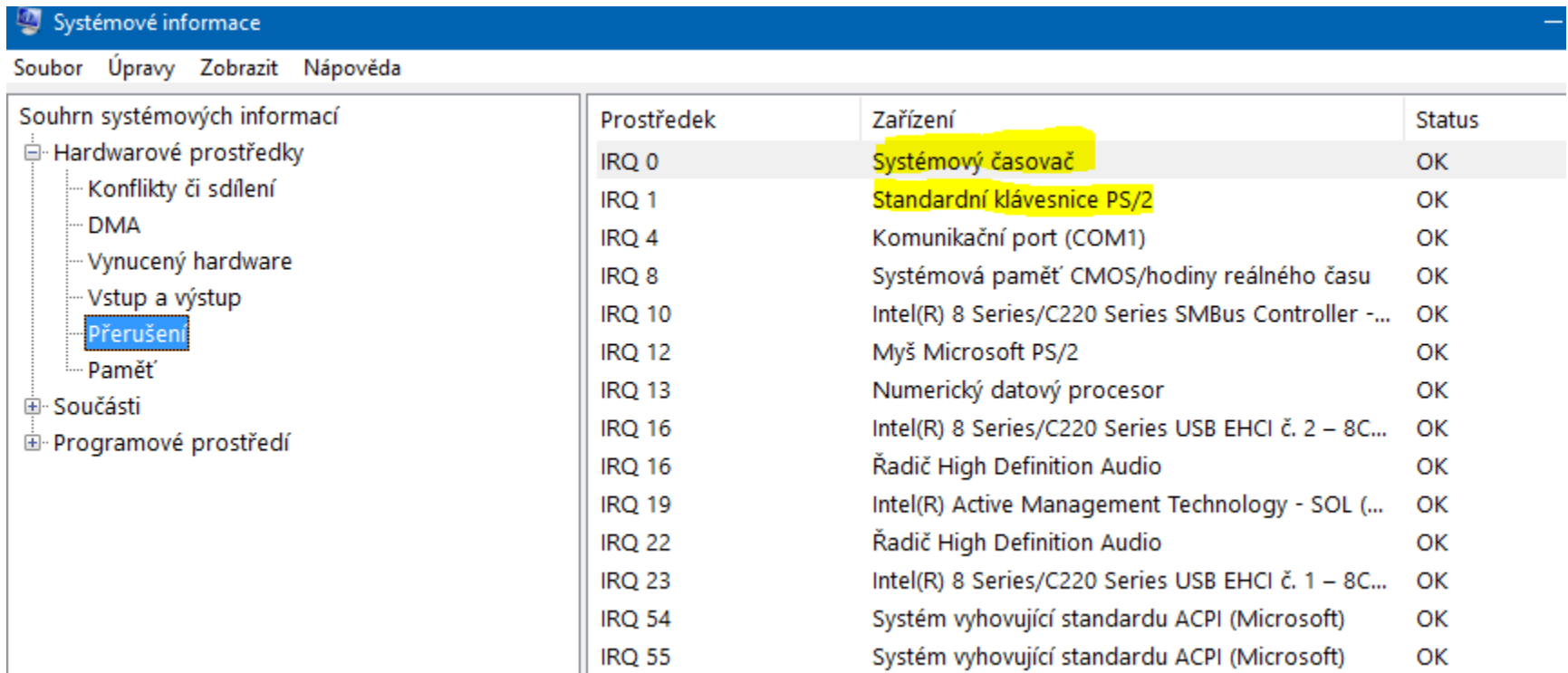
1. spravuje IRQ 0 až 7
(master, na IRQ2 je připojen druhý)

2. spravuje IRQ 8 až 16

novější systémy Intel APIC Architecture (24 IRQ)

Po datové sběrnici
řekne řadič
přerušení
procesoru, jaký
přerušovací vektor
(index do IDT)
použijeme

IRQ pod Win: msinfo32.exe

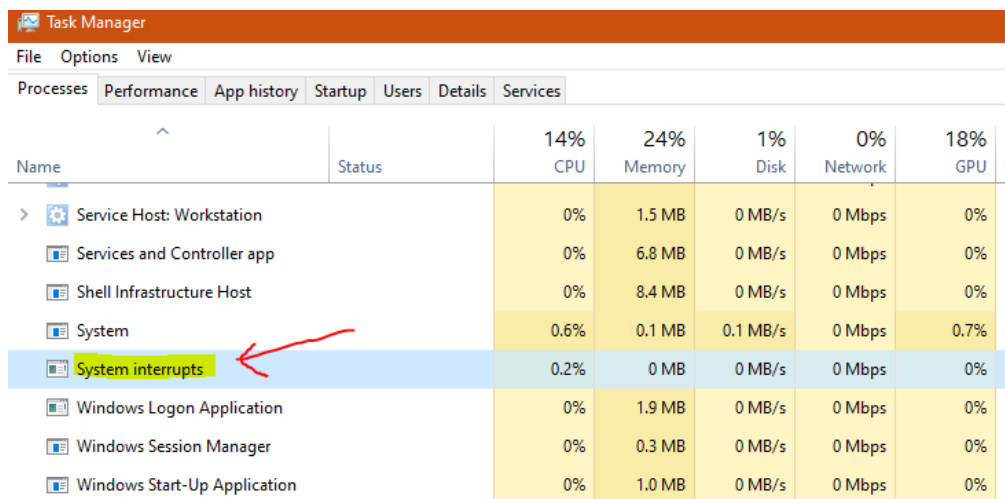


Prostředek	Zařízení	Status
IRQ 0	Systémový časovač	OK
IRQ 1	Standardní klávesnice PS/2	OK
IRQ 4	Komunikační port (COM1)	OK
IRQ 8	Systémová paměť CMOS/hodiny reálného času	OK
IRQ 10	Intel(R) 8 Series/C220 Series SMBus Controller - ...	OK
IRQ 12	Myš Microsoft PS/2	OK
IRQ 13	Numerický datový procesor	OK
IRQ 16	Intel(R) 8 Series/C220 Series USB EHCI č. 2 – 8C...	OK
IRQ 16	Řadič High Definition Audio	OK
IRQ 19	Intel(R) Active Management Technology - SOL (...)	OK
IRQ 22	Řadič High Definition Audio	OK
IRQ 23	Intel(R) 8 Series/C220 Series USB EHCI č. 1 – 8C...	OK
IRQ 54	Systém vyhovující standardu ACPI (Microsoft)	OK
IRQ 55	Systém vyhovující standardu ACPI (Microsoft)	OK

Linux: `cat /proc/interrupts`

Windows

Ve správci úloh vidíme položku System Interrupts.



The screenshot shows the Windows Task Manager Performance tab. The 'Processes' section is expanded, and the 'System Interrupts' process is highlighted in blue. A red arrow points to the 'System Interrupts' entry. The table below shows the resource usage for various system processes.

Name	Status	14% CPU	24% Memory	1% Disk	0% Network	18% GPU
> Service Host: Workstation		0%	1.5 MB	0 MB/s	0 Mbps	0%
Services and Controller app		0%	6.8 MB	0 MB/s	0 Mbps	0%
Shell Infrastructure Host		0%	8.4 MB	0 MB/s	0 Mbps	0%
System		0.6%	0.1 MB	0.1 MB/s	0 Mbps	0.7%
System Interrupts		0.2%	0 MB	0 MB/s	0 Mbps	0%
Windows Logon Application		0%	1.9 MB	0 MB/s	0 Mbps	0%
Windows Session Manager		0%	0.3 MB	0 MB/s	0 Mbps	0%
Windows Start-Up Application		0%	1.0 MB	0 MB/s	0 Mbps	0%

Souhrnný zástupný symbol používaný k zobrazení systémových prostředků používaných všemi hardwarovými přerušeními v počítači.

(citace viz <https://cs.phhsnews.com/what-is-system-interrupts-process-and-why-is-it-running-on-my-pc4743>)

Sdílení IRQ více zařízeními

- na jedno IRQ lze registrovat několik obslužných rutin (registrovány při inicializaci ovladače)
- do tabulky vektorů přerušení je zavěšena „**superobsluha**“
- superobsluha pouští postupně jednotlivé zaregistrované obsluhy, až jedna z nich zafunguje
- pokud dané přerušení vyvoláno najednou více zařízeními – zavolá opakovaně

dvě části ovladače zařízení

- první část

ve vlastním režimu obsluhy přerušení
musí trvat velmi krátkou dobu (stabilita)

- odložená část

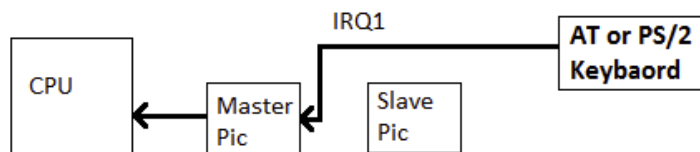
může naplánovat další část, která se vykoná „až bude čas“

DMA – přímý přístup do paměti

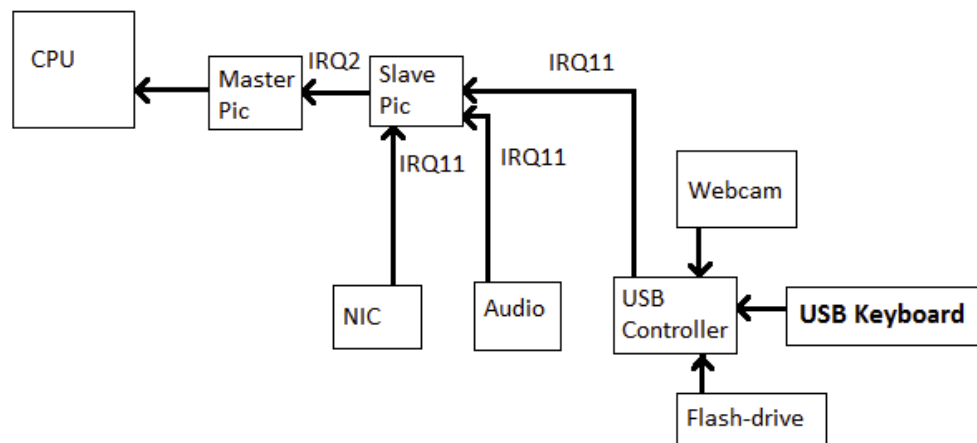
- Mechanismus umožňující perifernímu zařízení číst či zapisovat z/do operační paměti počítače bez účasti procesoru.
- Procesor dá pouze pokyn co se má provést (pomocí IN, OUT instrukcí naprogramuje řadič) a je informován o výsledku (pomocí IRQ signálu, tedy přerušením).
- Např. načtení stránky paměti ze swapu na disku do RAM a naopak
- Počáteční adresa zdroje
- Počáteční adresa cíle
- Počet přenášených dat (bytů)
- Přečtěte si: <https://cs.wikipedia.org/wiki/DMA>

Jak je to s dnešní USB klávesnicí?

Here is a simple visualization of the difference between an AT or PS/2 keyboard and a USB keyboard:



Starší klávesnice
samostatný konektor na PC
na připojení klávesnice

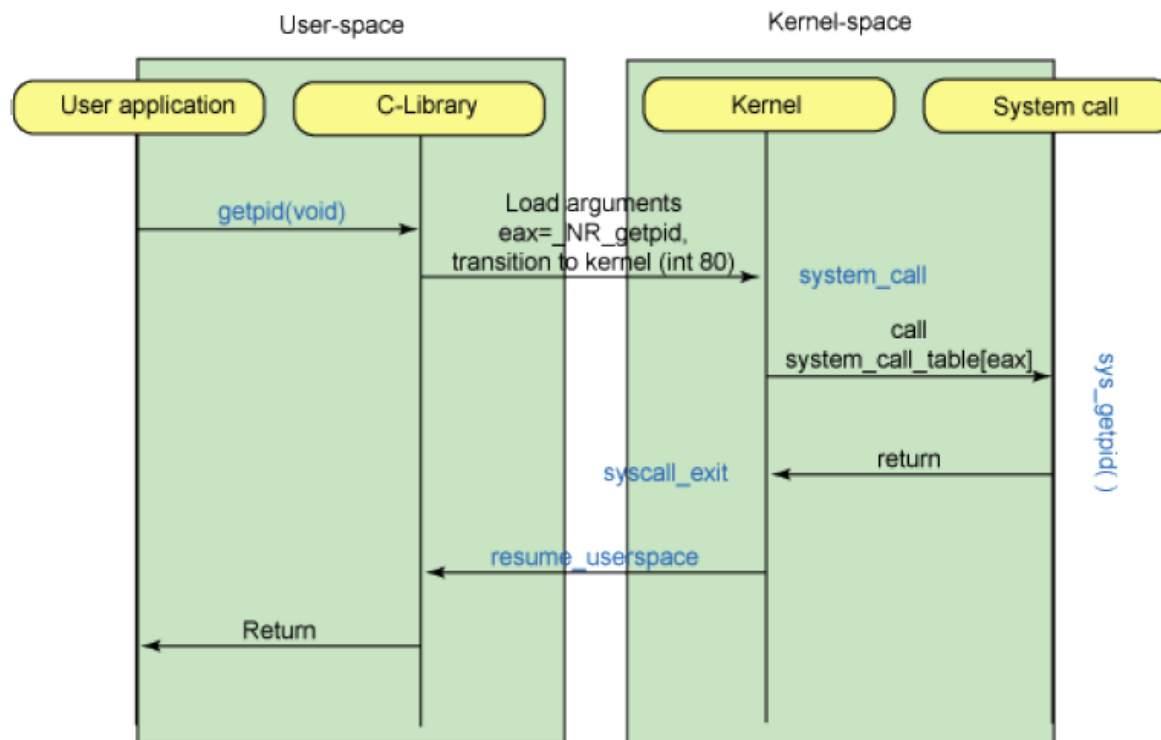


Dnešní USB klávesnice

Zdroj – doporučuji přečíst:
<http://superuser.com/questions/456459/computer-architecture-are-usb-keyboards-less-responsive-due-to-narrow-irq-range>

Jak probíhá volání systému např. při getpid() ?

Figure 1. The simplified flow of a system call using the interrupt method



Literatura - Podrobnosti

http://www.microbe.cz/docs/Frantisek_Rysanek-Routing_preruseni_a_kolize_prostredku_na_platforme_x86.pdf

Koncepce OS

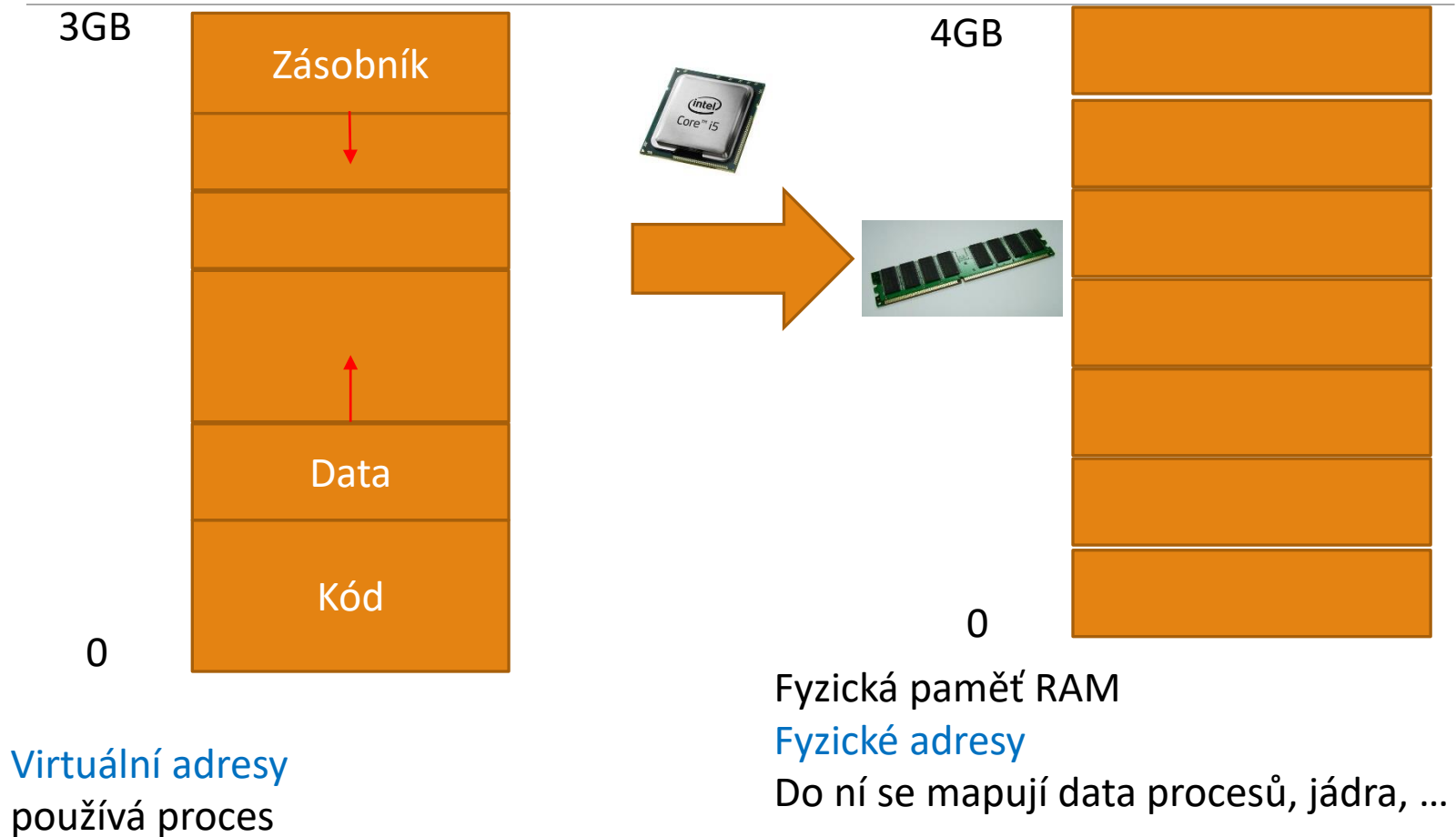
Základní abstrakce:

- procesy
- soubory
- uživatelská rozhraní

Procesy

- **Proces** – instance běžícího programu
- **Adresní prostor** procesu
 - Proces používá typicky **virtuální paměť** (od 0 do nějaké adresy), která se mapuje do **fyzické paměti** (RAM – paměťové čipy)
 - **MMU (Memory Management Unit)** zajišťuje mapování a tedy i soukromí procesu – je součástí procesoru
 - **kód** spustitelného programu, **data**, **zásobník**
- S procesem sdruženy **registry** a další info potřebné k běhu procesu = **stavové informace**
 - **registry** – čítač instrukcí **PC**, ukazatel zásobníku **SP**, univerzální registry

Adresní prostor procesu



Paměť RAM

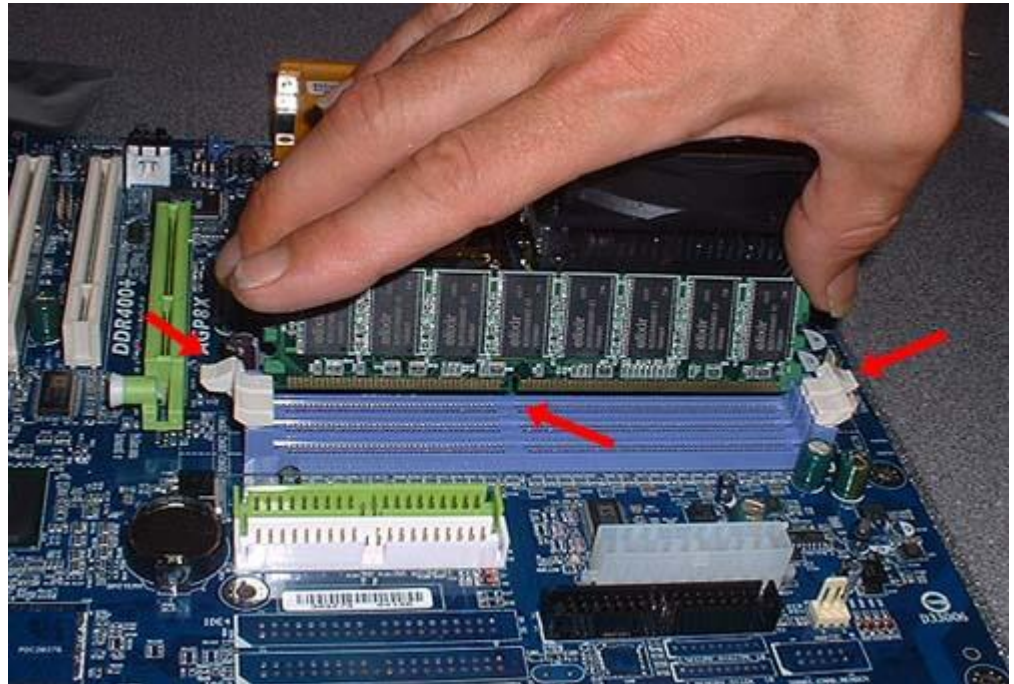
Fyzická operační paměť RAM

Při vypnutí napájení ztratí svůj obsah

Tvořena paměťovými chipy

Typická velikost RAM v dnešních PC a NB je:

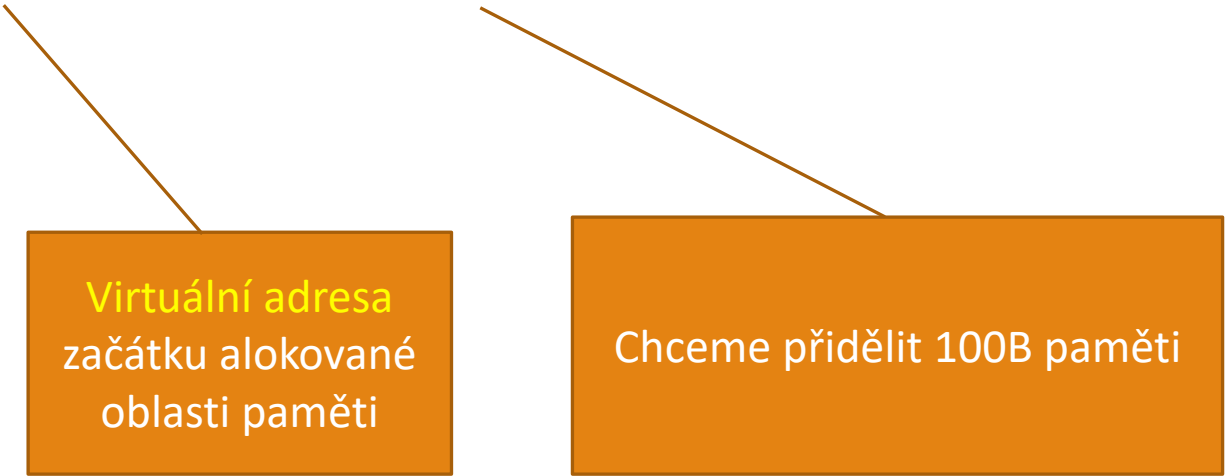
- 4 GB
- 8 GB
- 16 GB , 32 GB, 64GB



Zdroj obrázku: <http://www.custom-build-computers.com/Fitting-PC-Ram.html>

Paměť – příklad alokace

```
ptr = malloc (100);
```



Virtuální adresa
začátku alokované
oblasti paměti

Chceme přidělit 100B paměti

Na haldě se alokuje 100B

Na začátek alokované oblasti odkazuje virtuální adresa ptr (nějaké číslo)

Oblast je mapována do fyzické paměti (RAM) od nějaké jiné fyzické adresy

Registry (příklad architektura x86)

malé úložiště dat uvnitř procesoru

obecné registry

- **EAX, EBX, ECX, EDX** .. jako 32ti bitové
 - **AX, BX, CX, DX** .. využití jako 16ti bitové (dolních 16)
 - **AL, AH** .. využití jako 8bitové

obecné registry - uložení offsetu

- **SP** .. offset adresy vrcholu **zásobníku (!)**
- **BP** .. pro práci se zásobníkem
- **SI** .. offset zdroje (source index)
- **DI** .. offset cíle (destination index)

Registry

segmentové registry

- CS code segment (kód)
- DS data segment (data)
- ES extra segment
- FS volně k dispozici
- GS volně k dispozici
- SS stack segment (zásobník)

CS:IP, SS:SP, DS:SI, ES:DI

Registry

speciální

- **IP** .. offset vykonávané instrukce vůči CS (CS:IP, případně označení CS:EIP)
poznámka: v obecné teorii OS používáme pojem PC (program counter),
zde tím již konkrétně myslíme CS:IP
- **FLAGS** .. zajímavé jsou jednotlivé bity
 - **IF** .. interrupt flag (přerušování zakázáno-povoleno)
 - **ZF** .. zero flag (je-li výsledek operace 0)
 - **OF, DF, TF, SF, AF, PF, CF**

bližší info např. http://cs.wikipedia.org/wiki/Registr_procesoru
jde nám o představu jaké registry a k jakému účelu jsou

Registry (x86-64)

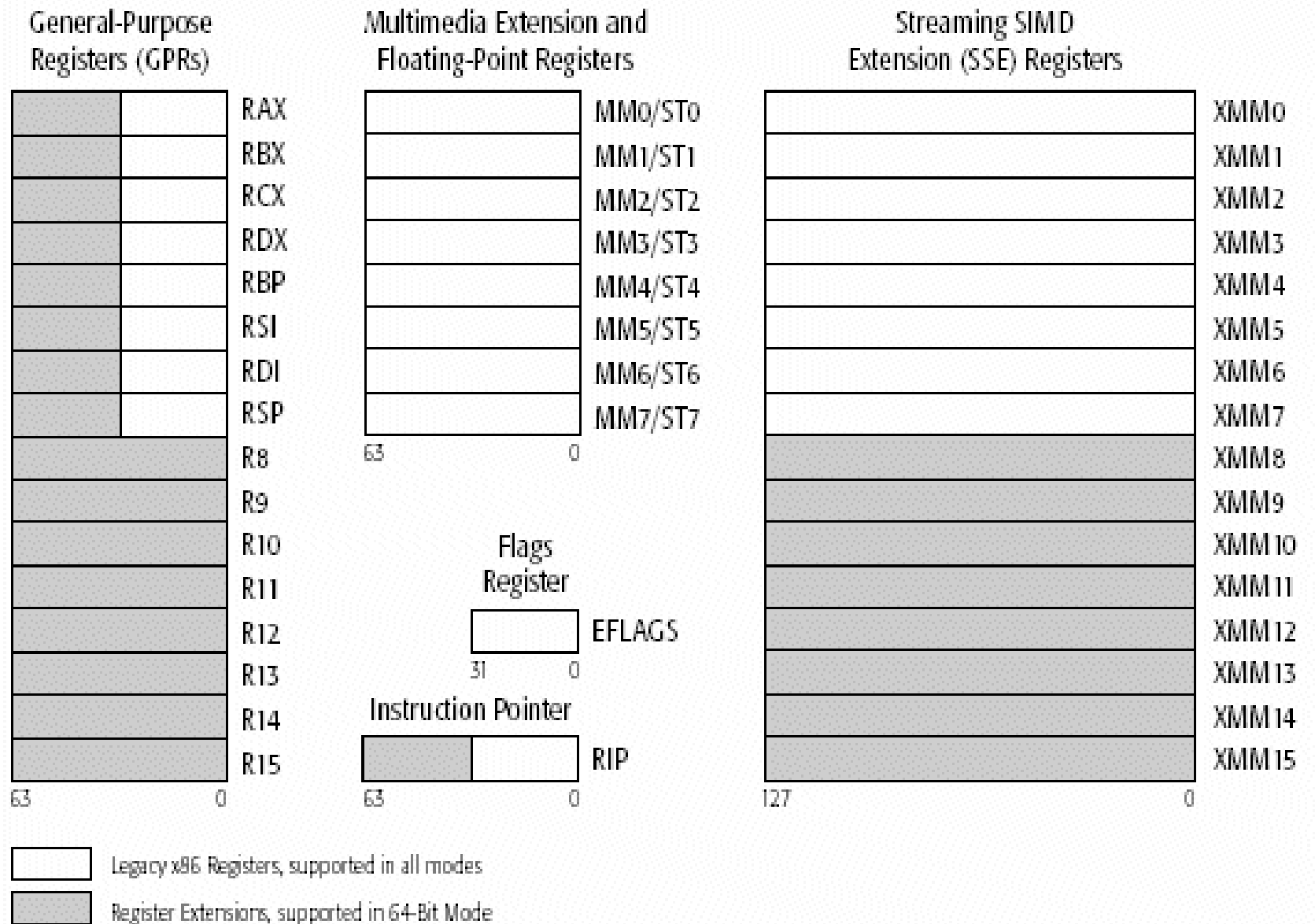
64bit



- For 16-bit operations, the two bytes of Register A are addressed as AX
- For 32-bit operations, the four bytes of Register A are addressed as EAX
- For 64-bit operations, the eight bytes of Register A are addressed as RAX

zdroj:

http://pctuning.tyden.cz/index2.php?option=com_content&task=view&id=7475&Itemid=28&pop=1&page=0



Základní služby OS pro práci s procesy

Vytvoření nového procesu

- `fork()` v UNIXu,
- `CreateProcess()` ve Windows

Ukončení procesu

- `exit()` v UNIXu,
- `ExitProcess()` ve Windows

Čekání na dokončení potomka

- `wait()`, `waitpid()` v UNIXu,
- `WaitForSingleObject()` ve Windows

Další služby - procesy

- **Alokace** a **uvolnění** paměti procesu
- **Komunikace** mezi procesy (IPC)
- **Identifikace** ve víceuživatelských systémech
 - identifikátor uživatele (**UID**)
 - skupina uživatele (**GID**)
 - proces běží s **UID** toho, kdo ho spustil (jsou i důležité výjimky)
 - v UNIXu – **UID**, **GID** – celá čísla
- Problém **uvíznutí** procesu

fork – systémové volání pro vytvoření procesu (!!!!)

- Vytvoří **identickou kopii** (klon) původního procesu
- Nový proces vykonává **stejný kód** (!!)
- Nový proces má jiný PID
- Návratová hodnota fork (!!)
 - rodič - hodnota větší než nula (PID potomka)
 - potomek – nula (signalizuje, že je potomek)
 - záporná hodnota, pokud nemůže proces vytvořit

UNIX a Linux

- Služba **fork()** – vytvoří přesnou kopii rodičovského procesu
- Návratová hodnota – rozliší mezi rodičem a potomkem (potomek dostane 0)
- Po forku běží dva procesy – původní rodič a nový potomek

```
pid = fork();
```

```
if (pid == 0)  
    { printf("potomek"); }  
else printf("rodič");
```

- potomek něco dělá a potom může činnost ukončit pomocí **exit()**
- rodič může na potomka čekat – **wait()**

fork – ukázka programu

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

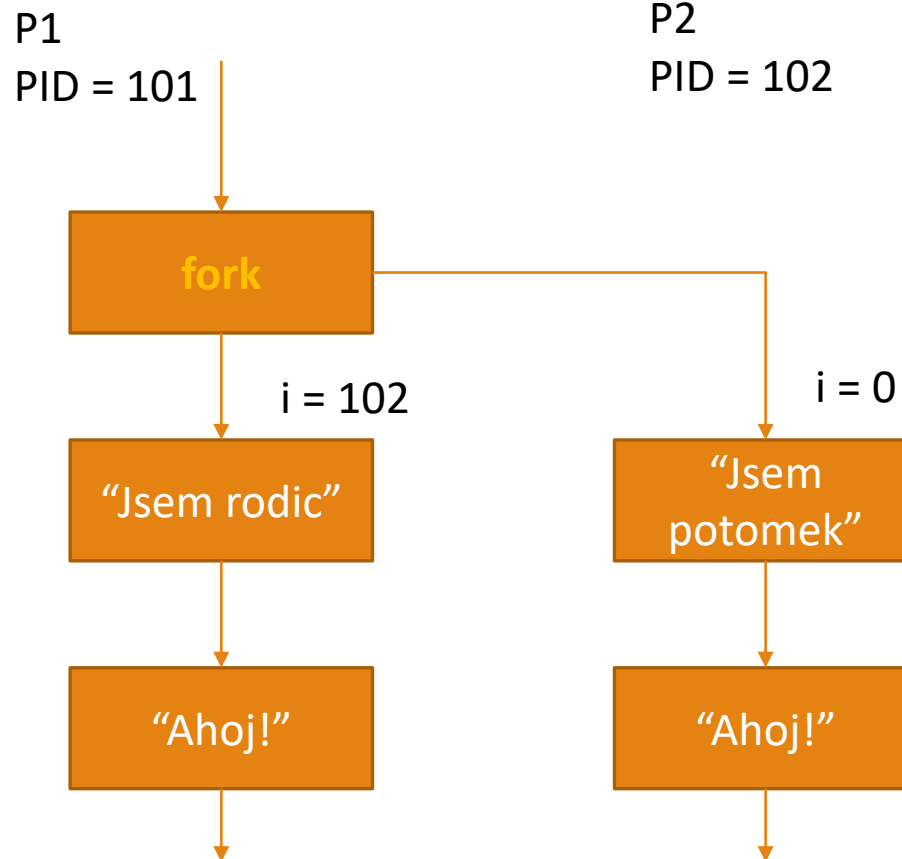
int main (void) {
    int i;
    i = fork(); //vytvoříme nový proces
    if (i == 0)
        printf ("Jsem potomek \n");
    else
        printf ("Jsem rodic \n");
    printf ("Ahoj! \n");
}
```

Vyzkoušejte na eryx.zcu.cz:

vim fork1.c

gcc -o fork1 fork1.c
./fork1

Graf procesů (!!)



Příklad 2

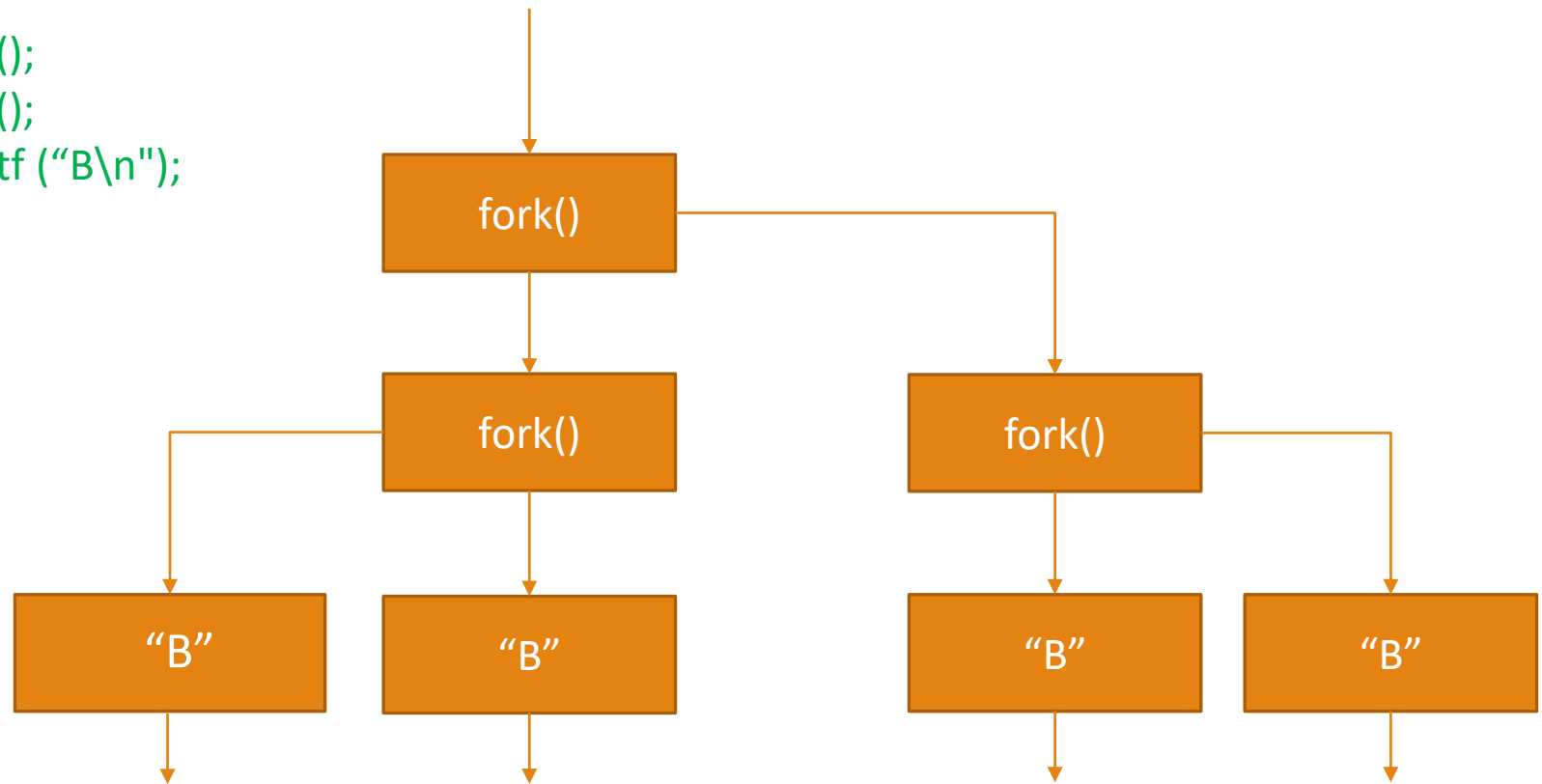
```
fork();  
fork();  
printf ("B \n");
```

Co bude
výstupem?

Nakreslete
graf

Řešení 2

```
fork();  
fork();  
printf ("B\n");
```



Příklad 3

```
if ( fork() == 0) {  
    fork();  
    printf ("A");  
}  
else  
    printf ("B");  
printf ("C");
```

Jak zařídit, aby proces vykonával jiný program?

- Systémové volání `execve()`
- Specifikujeme, **jaký program** má náš proces začít vykonávat
- Vykonává jiný kód
- PID a vazba na rodiče zůstane

UNIX – execve

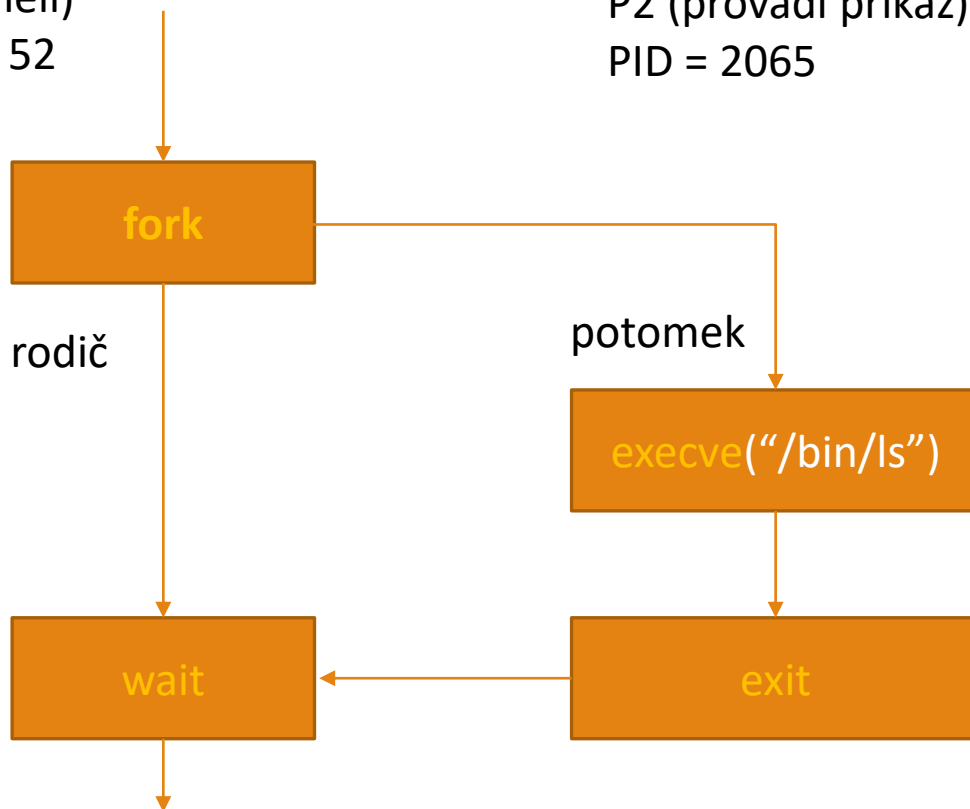
Potomek může místo sebe spustit jiný program – volání `execve()` – nahradí obsah paměti procesem spouštěným ze zadaného souboru

1. `if (fork() == 0)`
2. `execve("/bin/ls", argv, envp);`
3. `else`
4. `wait(NULL);`

Graf procesů

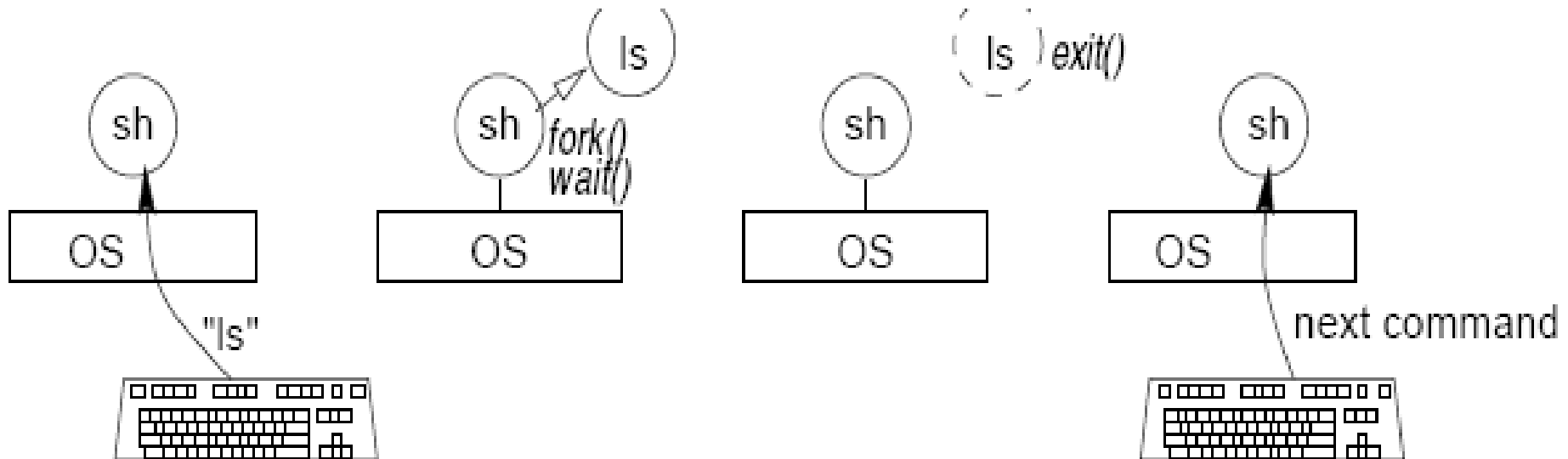
P1 (shell)
PID = 52

P2 (provádí příkaz)
PID = 2065



Příkazový interpret

Spouští příkaz – vytvoří nový proces, čeká na jeho dokončení; ukončení – volání `sl. systému`



Windows (Win32)

- Vytvoření procesu službou `CreateProcess()`
- Vytvoří nový proces, který vykonává program zadaný jako parametr
- Mnoho parametrů – vlastnosti procesu

Ukázka pod Windows

```
STARTUPINFO StartInfo; // name structure
PROCESS_INFORMATION ProcInfo; // name structure
memset(&ProcInfo, 0, sizeof(ProcInfo)); // Set up memory block
memset(&StartInfo, 0, sizeof(StartInfo)); // Set up memory block
StartInfo.cb = sizeof(StartInfo); // Set structure size
int res = CreateProcess(NULL, "MyApp.exe", NULL, NULL, NULL, NULL, NULL, NULL, &StartInfo, &ProcInfo); // starts
MyApp
if (res)
{
    WaitForSingleObject(ProcInfo.hThread, INFINITE); // wait forever for process to finish
    SetFocus(); // Bring back focus
}
```

příklad viz

<http://msdn.microsoft.com/en-us/library/windows/desktop/ms682512%28v=vs.85%29.aspx>

Soubory

- Zakrytí podrobností o discích a I/O zařízení
- Poskytnutí abstrakce – soubor
- Systémová volání
 - vytvoření, zrušení, čtení, zápis
- Otevření a uzavření souboru – open, close
- Sekvenční nebo náhodný přístup k datům - lseek
- Logické sdružování souborů do adresářů
- Hierarchie adresářů – stromová struktura (narušená linky)

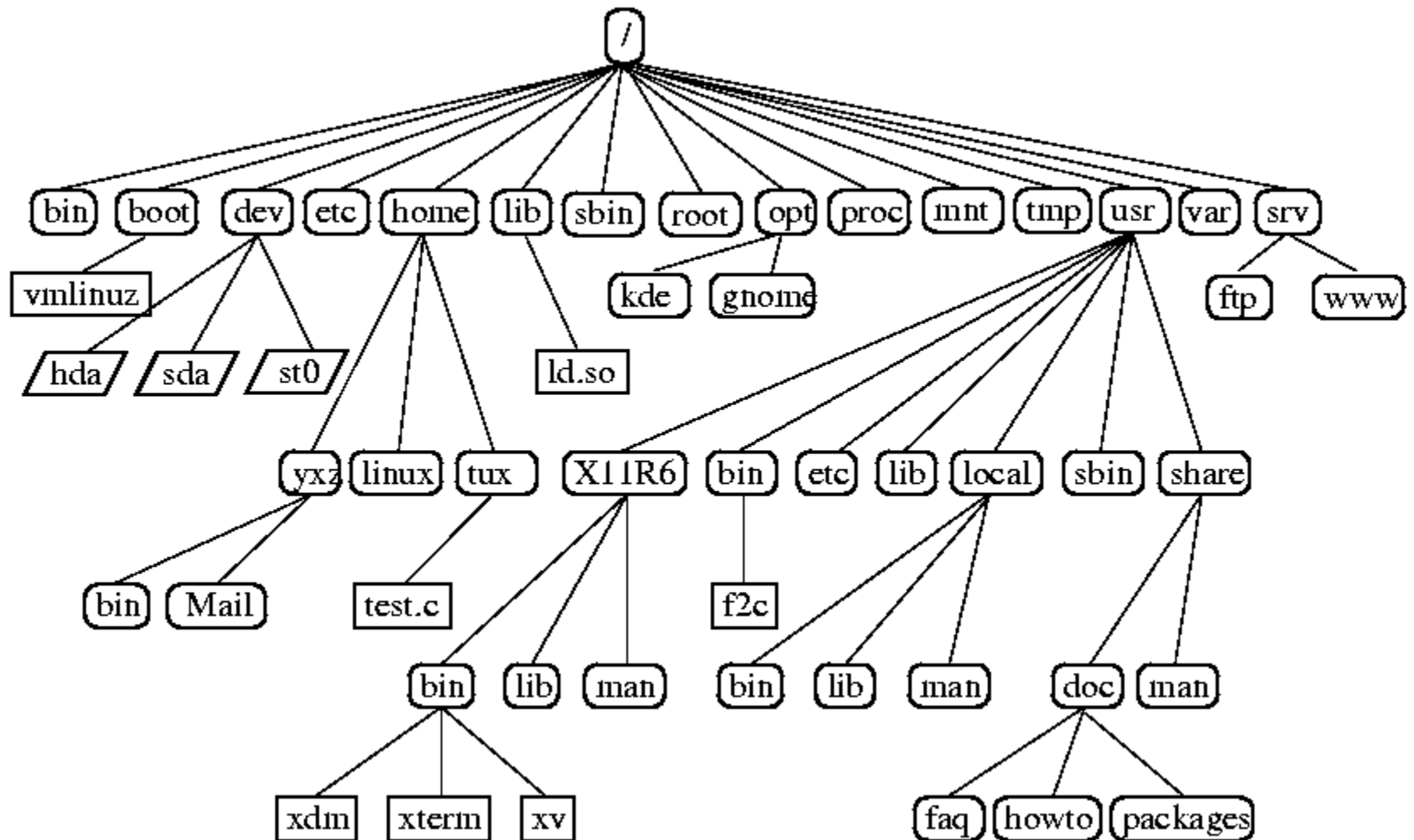
Soubory II.

Ochrana souborů, adresářů **přístupovými právy**

- kontrola při otevření souboru
- pokud není přístup – chyba

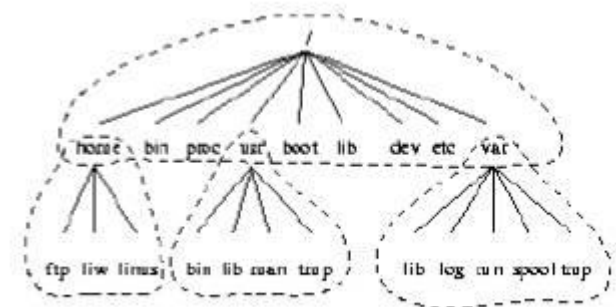
Připojitelnost souborových systémů

- Windows – disk určený prefixem **C:**, **D:**
- Unix – *připojení v rámci* adresářového stromu



Linux – filesystem

zdroj: <http://www-uxsup.csx.cam.ac.uk/pub/doc/suse/suse9.0/userguide-9.0/ch24s02.html>
<http://www.cs.wits.ac.za/~adi/courses/linuxadmin/content/module2doc.html>



Sekvenční x náhodný přístup

Sekvenční přístup

- soubor musíme číst od začátku do konce
- nemůžeme přeskakovat, vracet se
- příkladem např. magnetická páska
- (lze rewind a znovu číst od začátku)

Náhodný přístup

- nejběžnější
- můžeme přeskakovat, vracet se libovolně
- potřebujeme operaci **lseek**

Uživatelské rozhraní

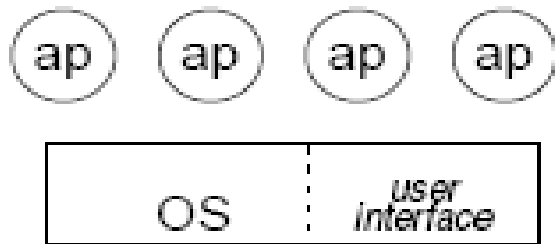
řádková – CLI (Command Line Interface)

grafická – GUI

původně UI součást jádra

v moderních OS – jedním z programů, možnost náhrady za jiné

UI – obrázky



UI jako součást jádra

UI v uživ. režimu

Kolik přepnutí kontextu je potřeba?
vs. vliv na stabilitu jádra OS

Uživatelské rozhraní - příklady

GUI Linux

- systém **XWindow** (zobrazování grafiky) a grafické prostředí (**správci oken**,...) – programy v uživatelském režimu

Windows XP

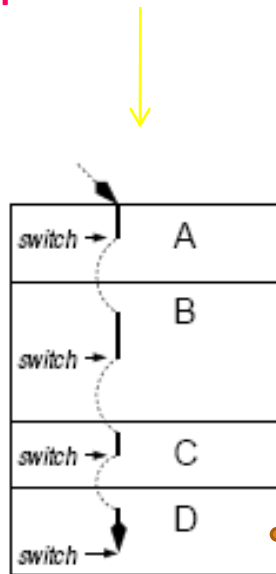
- grafická část v jádře
- logická část (v uživatelském režimu)
- výkon vs. stabilita

Proces jako abstrakce

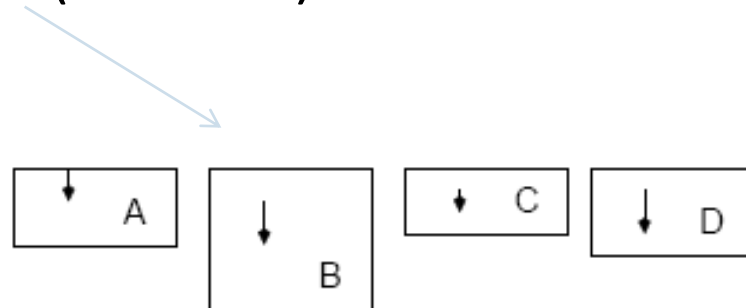
- **Proces** – běžící program včetně obsahu čítače instrukcí, registrů, proměnných; běží ve vlastní paměti
- Konceptně každý proces – vlastní **virtuální** CPU
- **Reálný** procesor – přepíná mezi procesy (multiprogramování)
- Představa množiny procesů běžících (pseudo)paralelně

Ukázka

4 procesy, každý má vlastní **bod běhu** (kam ukazuje čítač instrukcí)
pseudoparalelní běh x **paralelní** (více CPU)



1 procesor




4 procesory

z pohledu uživatele se nám jeví
jako paralelní vykonávání procesů

Pseudoparalelní běh

- Pseudoparalelní běh – v jednu chvíli aktivní pouze **jeden** proces (při 1 CPU)

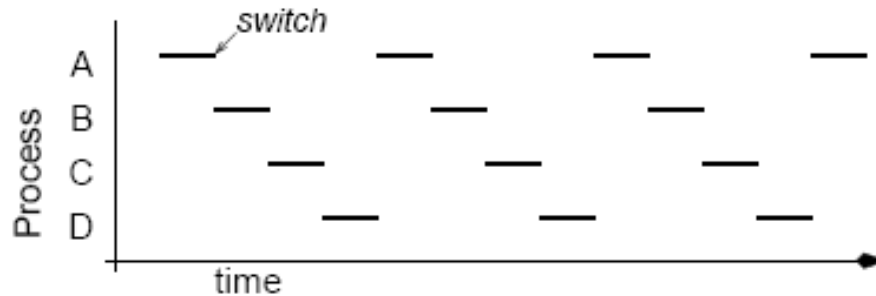
- Po určité době **pozastaven** a spuštěn další • • •



Vyprší časové kvantum,
nebo chce I/O operaci

- Po určité době všechny procesy vykonají část své činnosti

Pseudoparalelní běh



Procesy A, B, C, D se střídají na procesoru
Z obrázku se zdá, že na procesoru stráví vždy stejnou dobu, ale nemusí tomu tak být – např. mohou požadovat I/O operaci a „odevzdají“ procesor dříve

Rychlost procesů

- Rychlost běhu procesu **není konstantní**.
- Obvykle **není** ani **reprodukovatelná**.
- Procesy nesmějí mít vestavěné **předpoklady o časování** (tj. že určitý úsek vykonání kódu trvá nějaký čas)
 - Např. doba trvání I/O různá.
 - Pokud potřebujeme něco načasovat -> využít časovač
- Procesy **neběží stejně rychle**.

Proces běží v reálném systému, který se věnuje i dalším procesům, obsluze přerušení atd., tedy nesmíme spoléhat, že poběží vždy stejně rychle..

Stavy procesu

- Procesy často potřebují **komunikovat** s ostatními procesy:

- *ls -l | more*


• • •

Oba jsou spuštěny
současně

- proces *ls* vypíše obsah adresáře na standardní výstup (deskriptor 1)
- *more* zobrazí text na jednu obrazovku a čeká na stisk klávesy
- Proces *more* je připraven běžet, ale **nemá** žádný **vstup** – **zablokuje se** dokud vstup nedostane

Kdy proces neběží – 2 možnosti

Blokování procesu – proces nemůže pokračovat, protože **čeká** na zdroj (vstup, zařízení, paměť), který není dostupný – proces nemůže **logicky** pokračovat, i když je CPU volné.



Nemůže,
na něco
čeká

Proces může být **připraven pokračovat**, ale CPU **vykonává jiný** proces – musí počkat, až bude CPU „volné“

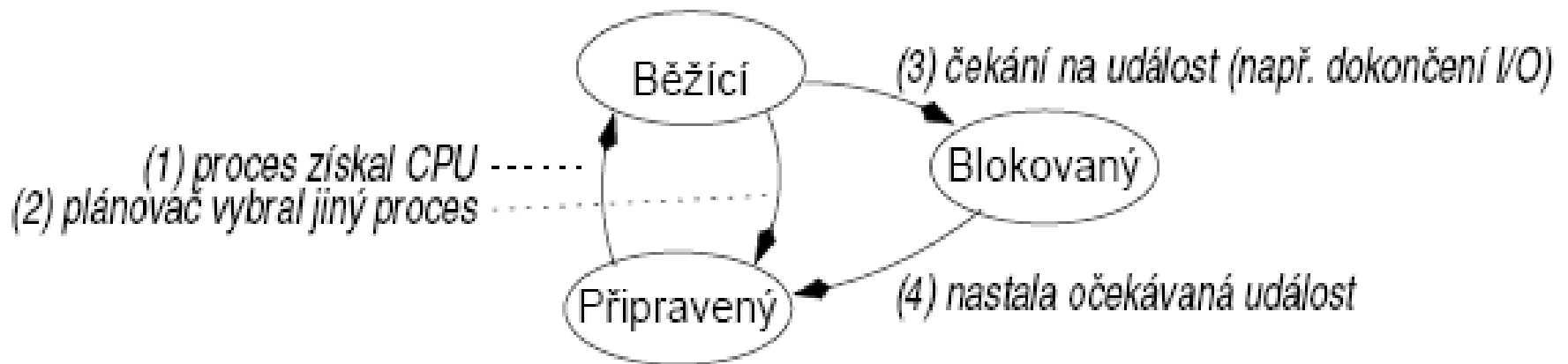


Chtěl by,
ale není
volný
CPU

Základní stavy procesu

- **Běžící** (running)
 - skutečně využívá CPU, vykonává instrukce
- **Připravený** (ready, runnable)
 - dočasně pozastaven, aby mohl jiný proces běžet
- **Blokovaný** (blocked, waiting)
 - neschopný běhu, dokud nenastane externí událost

Základní stavy procesu (!!)



Nově vytvořený proces jde do stavu Připravený

Přechody stavů procesu

1. Plánovač **vybere** nějaký proces
 2. Proces je **pozastaven**, plánovač vybere jiný proces (typicky - vypršelo časové kvantum u interaktivních systémů)
 3. Proces se **zablokuje**, protože čeká na událost (na nějaký zdroj – disk, čtení z klávesnice)
 4. **Nastala** očekávaná **událost**, např. zdroj se stal dostupný
-
- Pokud je proces **nově vytvořený**, jde do stavu Připravený. Až plánovač jej vybere a přidělí mu procesor.
 - Když se blokovaný proces dočká události, jde do stavu Připravený.
 - Kdo dostane CPU, **vždy vybírá plánovač** ze stavu Připravený.
 - Ze stavu Běžící si můžeme představit šipku do stavu Ukončený. (resp. do stavu Zombie a následně Ukončený)

Stavy procesů

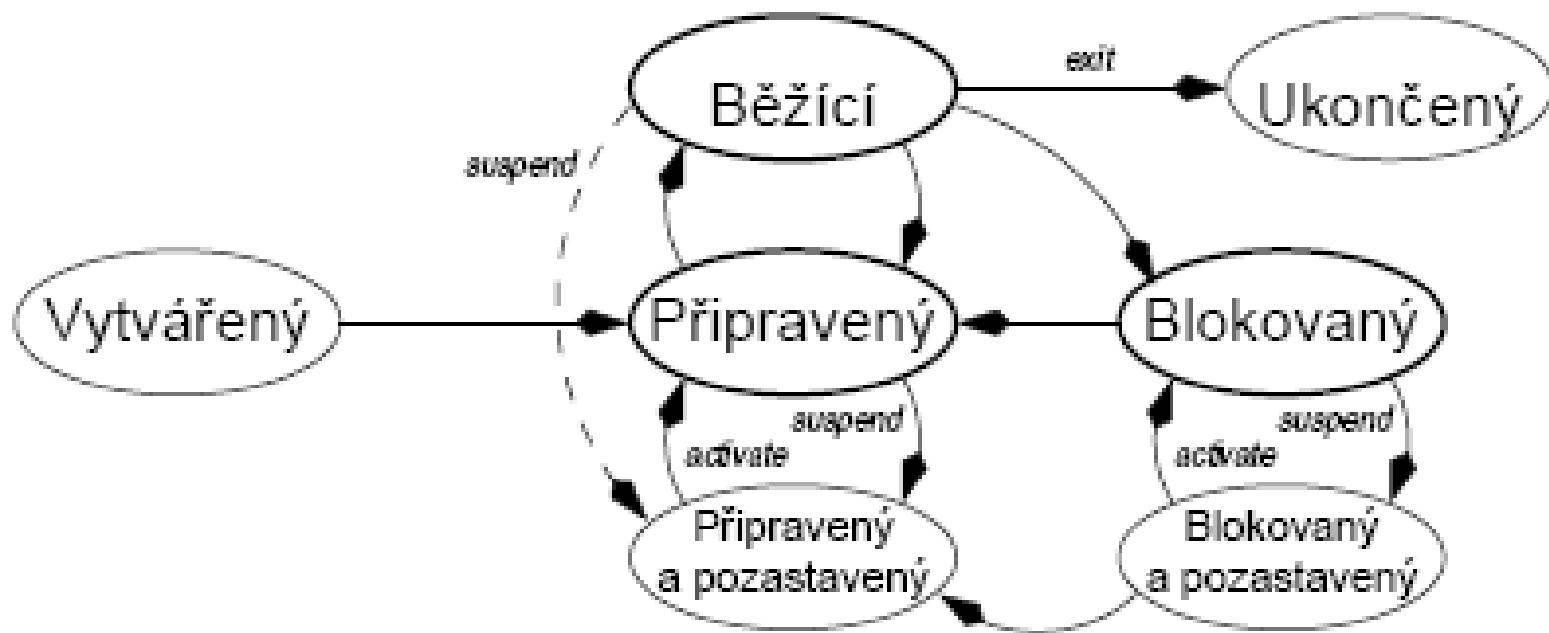
- Jádru OS obsahuje plánovač
- Plánovač určuje, který proces bude běžet
- Nad OS řada procesů, střídají se o CPU

Stav procesu **pozastavený**

- V některých systémech může být proces **pozastaven** nebo **aktivován**
- V diagramu přibudou **dva** nové stavy

V teorii často mluvíme o plánování procesů, v běžných dnešních systémech se pak plánují jednotlivá vlákna.

Stavy procesů



Pozastavené procesy nesoupeří o čas CPU, snižují aktuální zatížení systému

Tabulka procesů

OS si musí vést evidenci, jaké procesy v systému v danou chvíli existují.

Tato informace je vedena v **tabulce procesů**.

Každý proces v ní má záznam, a tento záznam se nazývá **process control block (PCB)**.

Na základě informací zde obsažených se plánovač umí rozhodnout, který proces dále poběží a bude schopen tento proces spustit ze stavu, v kterém byl naposledy přerušen.

PCB (Process Control Block) !

- OS udržuje tabulku nazývanou **tabulka procesů**
- Každý proces v ní má položku zvanou **PCB** (Process Control Block)
- PCB obsahuje všechny informace potřebné pro **opětovné spuštění** přerušného procesu
 - **Procesy se o CPU střídají, tj. běh procesu je přerušovaný**
- Konkrétní obsah PCB – různý dle OS
- Pole správy **procesů**, správy **paměti**, správy **souborů** (!!)

Položky - správa procesů

Identifikátory (číselné)

- Identifikátor procesu - PID
- Identifikátor uživatele - UID

Stavová informace procesoru

- Univerzální registry,
- Ukazatel na další vykonávanou instrukci
-obecně název Program Counter - **PC - (konkrétně CS:IP)**
- ukazatel zásobníku SP (konkrétně SS:SP)
- **Stav CPU** – PSW (Program Status Word) – registr Flags

Stav procesu (běžící, připraven, blokován)

Plánovací parametry procesu (algoritmus, priorita)

Položky – správa procesů II

Odkazy na rodiče a potomky

Účtovací informace

- Čas spuštění procesu
- Čas CPU spotřebovaný procesem

Nastavení meziprocesové **komunikace**

- Nastavení signálů, zpráv

Položky – správa paměti

Popis paměti

- Ukazatel, velikost, přístupová práva

1. Úsek paměti s kódem programu

2. Data – hromada

- Pascal – new release
- C – malloc, free

3. Zásobník

- Návrátové adresy, parametry funkcí a procedur, lokální proměnné funkcí

Položky – správa souborů

Nastavení prostředí

- Aktuální pracovní adresář

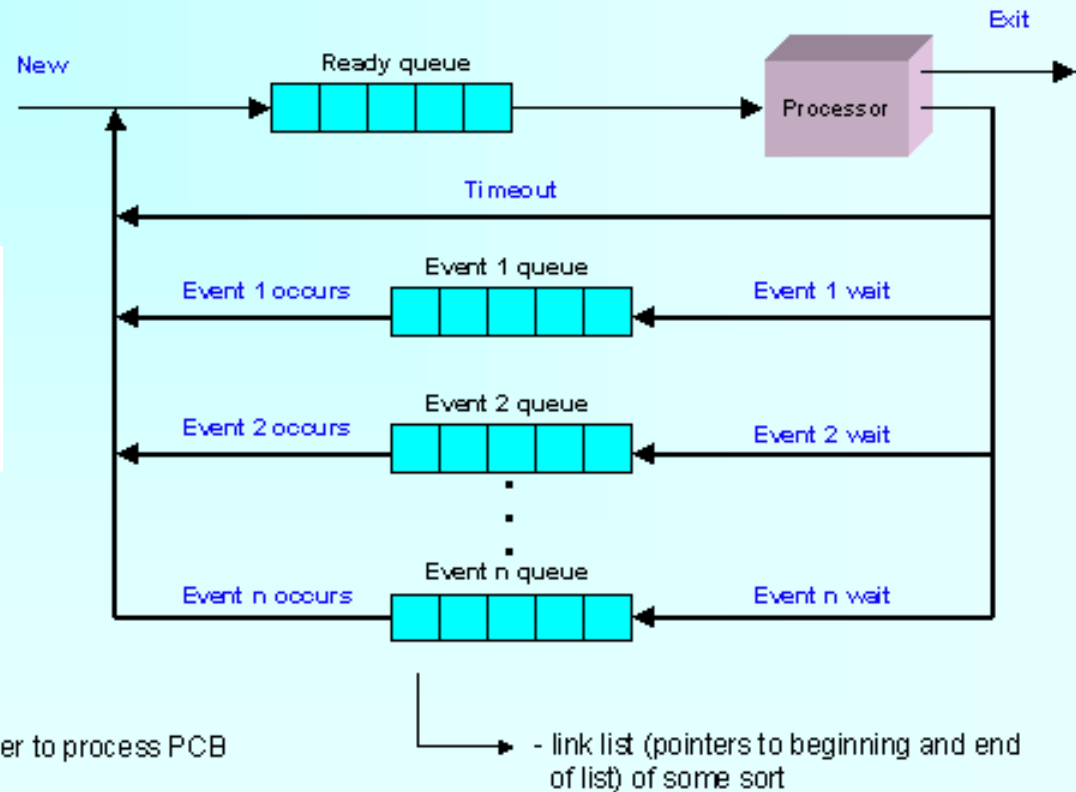
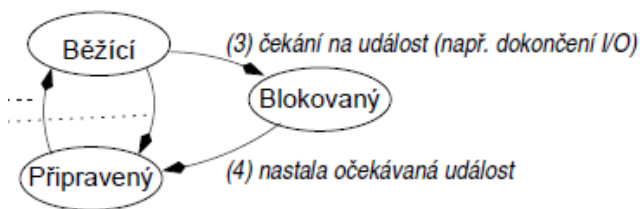
Otevřené soubory

- Způsob otevření – čtení / zápis
- Pozice v otevřeném souboru

PCB

Pointer	Process state
Process number	
Program counter	
Registers	
Memory limits	
List of open files	
...	

Data Structures (again)...



Yair Amir

Fall 00 / Lecture 2

8

Viz <http://www.cs.jhu.edu/~yairamir/cs418/os2/sld007.htm>

Poznámky

Stav **Nový**

- Proces přejde z nový do stavu Připravený

Stav **Ukončený**

- Přejod ze stavu běžící do ukončený, např. voláním exit

Častou chybou je, že lidé kreslí přechod ze stavu Nový do stavu Běžící, napřed se musí jít přes Připravený !
Stejně tak, do stavu Ukončený jdeme ze stavu Běžící.

Ukončení procesu - možnosti

- I. Proces úspěšně vykoná kód programu 😊
- II. Proces překročí limit nějakého zdroje/výjimka (celočíselné dělení nulou)
- III. Proces ukončí uživatel (např. příkazem kill)

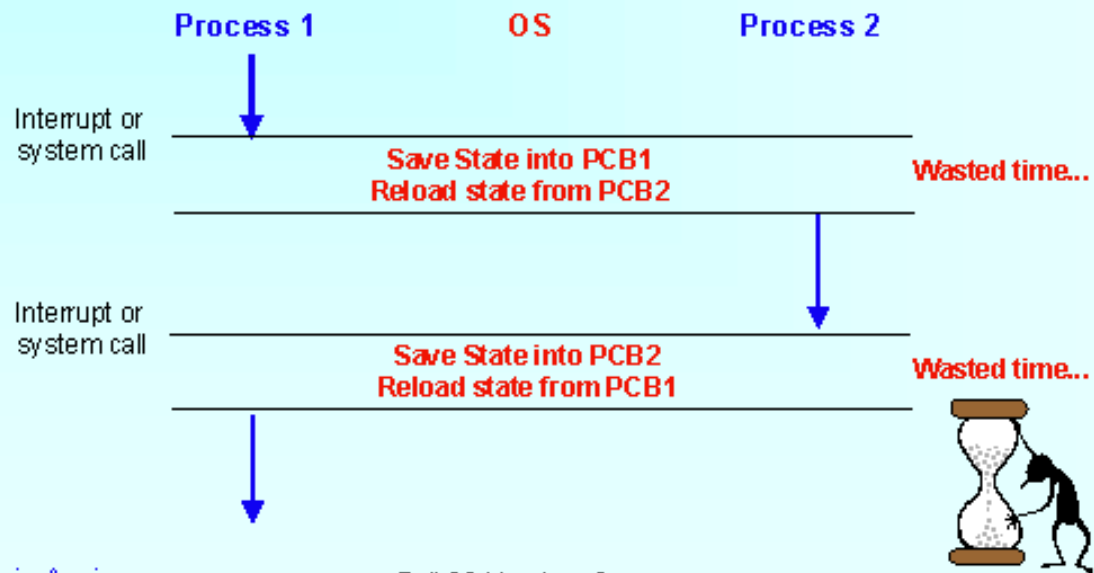
Pokud skončí rodičovský proces

- některé OS nedovolí pokračovat child procesu
- v Linuxu potomka adoptuje proces init a běží dále

Přepnutí procesu

Context Switch

Switching the CPU to another process requires saving the state of the old process and loading the saved state for the new process.



Yair Amir

Fall 00 / Lecture 2

10

Přepnutí procesu - průběh

- Systém použije HW přerušení od časovače (např. interaktivní systémy)
- Pravidelná přerušení
 - Zkontroluje, zda má proces ještě čas běžet
 - Pokud ano, návrat z přerušení a neděje se nic
 - Pokud ne, dojde k přeplánování na jiný proces – přepnutí kontextu
- U některých systémů, které nejsou založené na časovém kvantu, tak není přepnutí kontextu odvozené od časovače – např. dávkové

K přeplánování procesu nedojde při každém tiku časovače, ale až, když jich je tolik, že vyprší časové kvantum

Přepnutí procesu - II

Přepnutí kontextu

- Uloží obsah registrů do zásobníku
- Plánovač nastaví proces, který opouští CPU jako ready
- Vybere nový proces pro spuštění
- Nastaví mapu paměti nového procesu
- Nastaví zásobník, načte z něj obsah registrů
- Proveď návrat z přerušení – IRET (do PC adresa ze zásobníku, hodnota registru FLAGS, přepne do uživatelského režimu)

Rychlost CPU vs. paměti

Cílem následující vsuvky je říci, že výkon systému může degradovat nejenom časté střídání procesů, protože se pořád musí **přepínat kontext**, ale i fakt, že informace v cachi se po přepnutí na jiný proces stane neaktuální, a cache paměti chvíli trvá, než se naplní aktuálními daty, což má také vliv na výkon systému.

Rychlost CPU vs. paměť

CPU

- Rychlost – počet instrukcí za sekundu
- Obvykle **nejrychlejší** komponenta v systému
- Skutečný počet instrukcí závisí na rychlosti, jak lze instrukce a data přenášet z a do hlavní paměti

Hlavní paměť

- Rychlost v paměťových cyklech (čtení, zápis)
- O řád pomalejší než CPU
- Proto důvod používat cache paměť

Rozdíly rychlostí – „pyramida“

- **CPU** registry – rychlé – zápisníková paměť, 32x32 nebo 64x64 bitů, žádné zpoždění při přístupu
- **Cache** – malá paměť s vysokou rychlostí,
 - princip lokality
(když data z adresy x , pravděpodobně budou potřeba i z $x+1$)
 - pokud jsou data v cache – dostaneme velmi rychle
- **RAM**
- **Vnější paměť**
 - Mechanická, pomalejší, větší kapacita, levnější cena za bit
 - Rotační disky, SSD disky, zálohovací média

MMU – Memory Management Unit

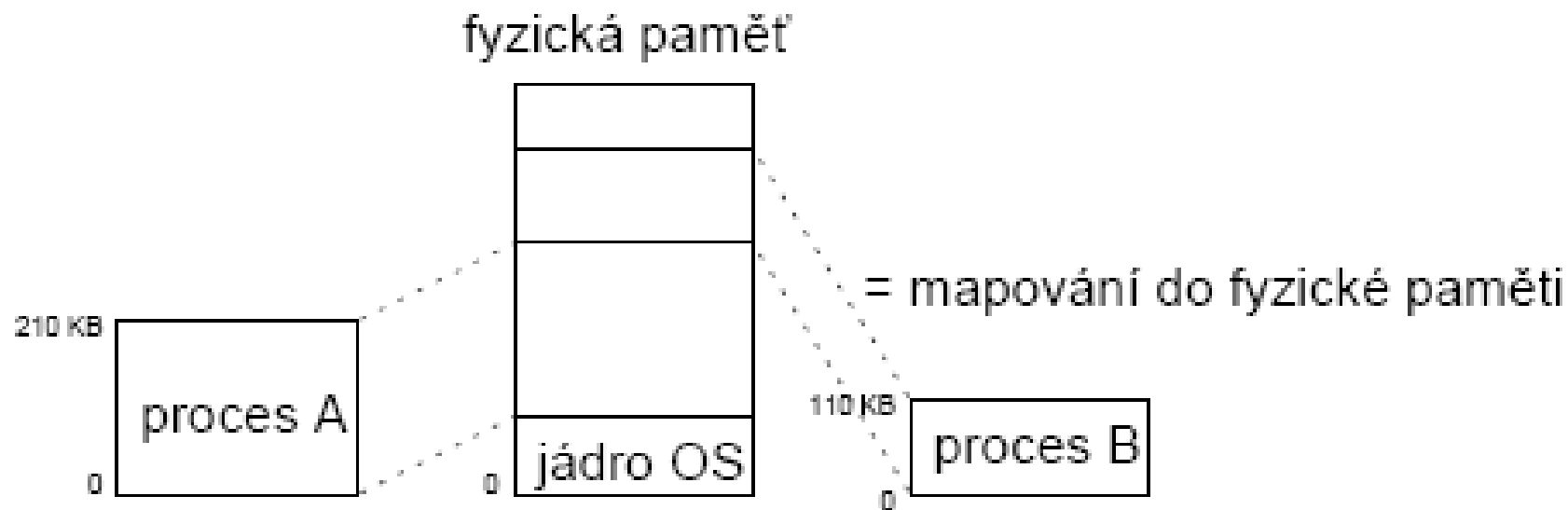
Více procesů v paměti

- Každý proces si myslí, že má paměť pro sebe, např. od adresy 0 (je potom třeba relokace na fyzickou adresu)
- Ochrana – nemůže zasahovat do paměti ostatních procesů ani jádra

Mezi CPU a pamětí je MMU

- Program pracuje s virtuálními adresami
- MMU je převede na fyzické adresy
- MMU je uvnitř procesoru, po které sběrnice jdou fyzické adresy

MMU – je uvnitř CPU



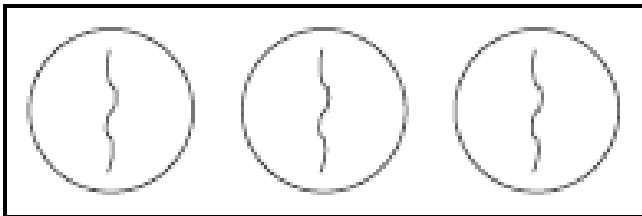
Výkonnostní důsledky

- Pokud program **nějakou dobu** běží
 - v cache **jeho** data a instrukce → dobrá výkonnost
- Při přepnutí na jiný proces
 - převažuje přístup do hlavní paměti (keš není naučená)
- Nastavení MMU se musí změnit
- Přepnutí mezi úlohami i přepnutí do jádra (volání služby OS) – relativně drahé (čas)

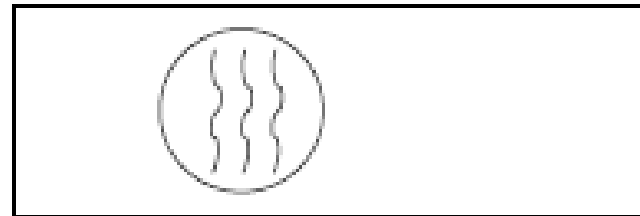
Procesy a vlákna

- Tradiční OS – každý proces má:
svůj vlastní adresový prostor
místo kde běží (**bod běhu**)
- Často výhodné – více bodů běhu, ale ve stejném adresovém prostoru
- Bod běhu – **vlákno** (thread, lightweight process)
- Více vláken ve stejném procesu - **multithreading**

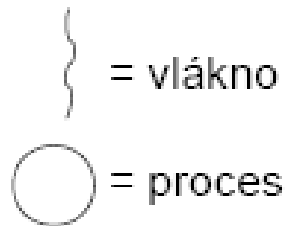
Procesy a vlákna



a) tradiční procesy



b) proces jako kontejner na vlákna



Vlákna (!!)

Vlákna v procesu **sdílejí (atributy procesu)**:
adresní prostor,
otevřené soubory

Vlákna **mají soukromý**:
čítač instrukcí,
obsah registrů,
soukromý zásobník
mohou mít soukromé **lokální** proměnné

Vlákna – příklady použití

Interaktivní procesy

- jedno vlákno pro komunikaci s uživatelem, další činnost na pozadí

Webový prohlížeč

- jedno vlákno příjem dat, další zobrazování a interakce s uživatelem

Textový procesor

- vlákno pro vstup dat, přeformátování textu, ...

Servery www i jiné

- jedno vlákno pro každého klienta
- např. v UPS jedno vlákno pro každou hru / klienta

Multithreading

- Podporován většinou OS
 - Linux, Windows
- Podporován programovacími jazyky
 - Java, knihovny v C, ...
- Proces začíná svůj běh s **jedním vláknem**, ostatní vytváří za běhu programově (konstrukce vytvoř vlákno)
- Režie na vytvoření vlákna a přepnutí kontextu menší než v případě procesů (!)

Pozn.: pokud přepínáme na vlákno jiného procesu, tak jde o přepnutí na jiný proces a režie je stejná, nižší je při přepínání mezi vlákny stejného procesu

Poznámka (terminologie)

Jeden proces – více vláken

- Ošetření souběžného přístupu ke sdílené paměti

Více procesů sdílejících paměť

- Ošetření souběžného přístupu ke sdílené paměti

V literatuře (např. při řešení synchronizace) se většinou nerozlišuje, zda uvažujeme souběžný přístup vláken nebo procesů ke společné paměti

Obojí umí způsobit problémy

Programové konstrukce pro vytváření vláken

Statické

- Proces obsahuje deklaraci pevné množiny podprocesů (např. tabulka)
- Všechny spuštěny při spuštění procesu

Dynamické

- Procesy mohou vytvářet potomky dynamicky
- Častější, s tím se spíše potkáme

Pro popis – [precedenční grafy](#)

Precedenční grafy

Process flow graph

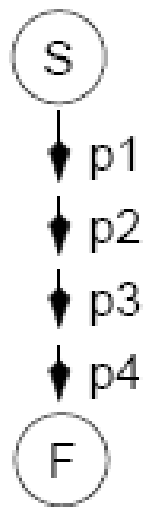
Popis pro vyjádření různých relací mezi procesy

Acyklický orientovaný graf

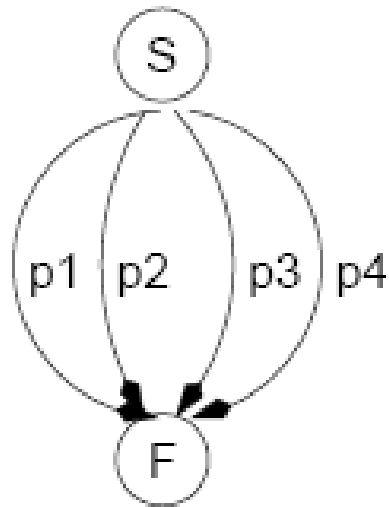
Běh procesu p_i – orientovaná hrana grafu

Vztahy mezi procesy – sériové nebo paralelní spojení – spojením hran

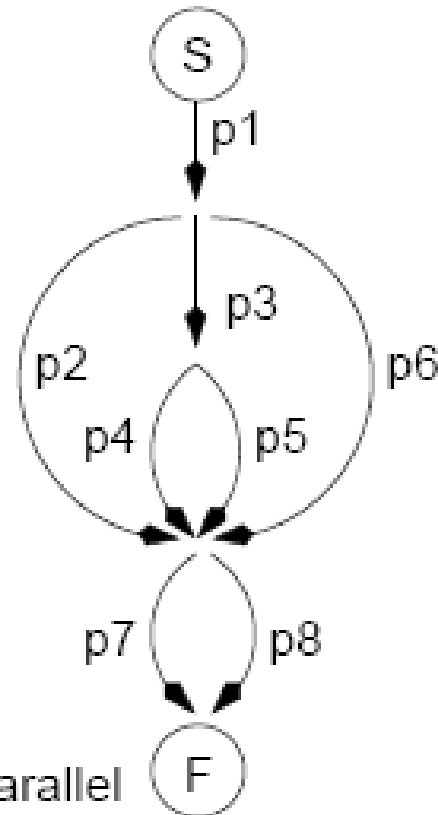
Precedenční grafy



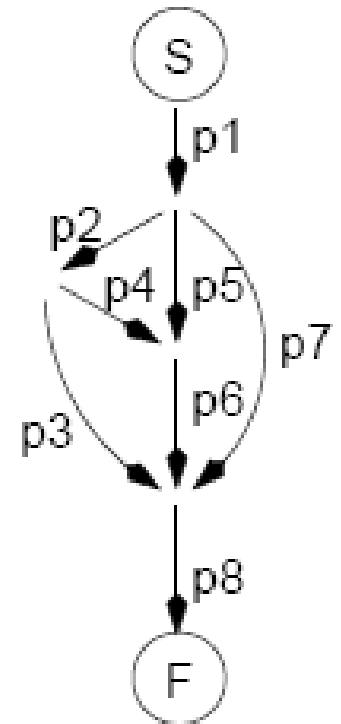
a) Series



(b) Parallel



(c) Series/Parallel



(d) General

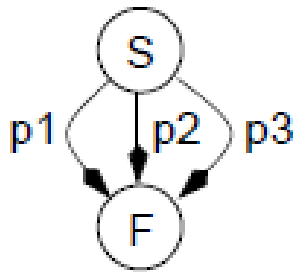
Fork, join, quit

Pozor neplést tento fork pro obecný popis se systémovým voláním fork

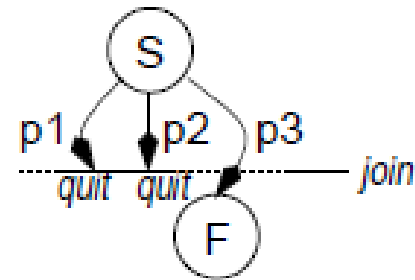
Mechanismus pro obecný popis paralelních aktivit

primitivum	funkce
fork X;	Spuštění nového vlákna od příkazu označeného návěstím X; nové vlákno poběží paralelně s původním
quit ;	Ukončí vlákno
join t, Y;	Atomicky (nedělitelně) provede: $t = t - 1$; if ($t == 0$) then goto Y;

Běh procesů odpovídající precedenčnímu grafu



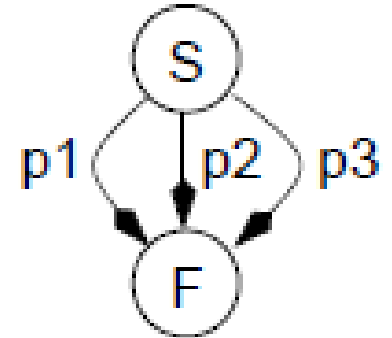
a) precedenční graf



b) skutečný běh

Nevíme, který z procesů doběhne první a který poslední,
jen jeden z možných běhů

Zápis pomocí fork-join-quit



n=3;

fork L2;

fork L3;

p1; join n, L4; quit;

L2: p2; join n, L4; quit;

L3: p3; join n, L4; quit;

L4:

F:

Poznámky k fork-join-quit

+ obecný zápis

- špatná čitelnost (přehlednost)

V některé literatuře se neuvádí quit, a předpokládá se $\text{join} = \text{join} + \text{quit}$

Správně vnořené precedenční grafy

$S(a,b)$ – sériové spojení procesů

(za procesem **a** následuje **b**)

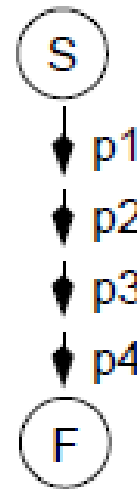
$P(a,b)$ – paralelní spojení procesů **a** a **b**

Precedenční graf je **správně vnořený**,
pokud může být popsán kompozicí
funkcí **S** a **P**

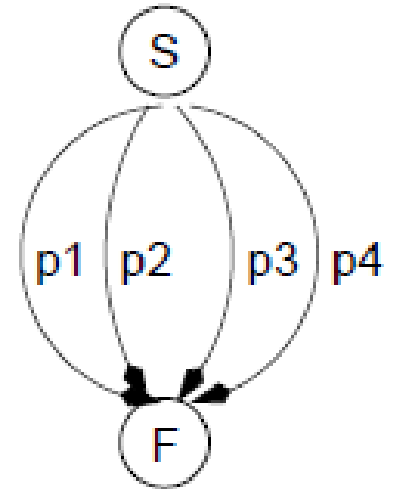
Příklady správně vnořených grafů

$S(p1, S(p2, S(p3, p4)))$

$P(p1, P(p2, P(p3, p4)))$



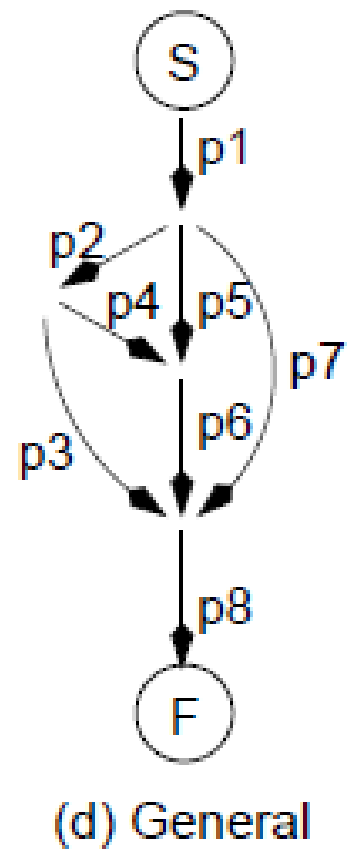
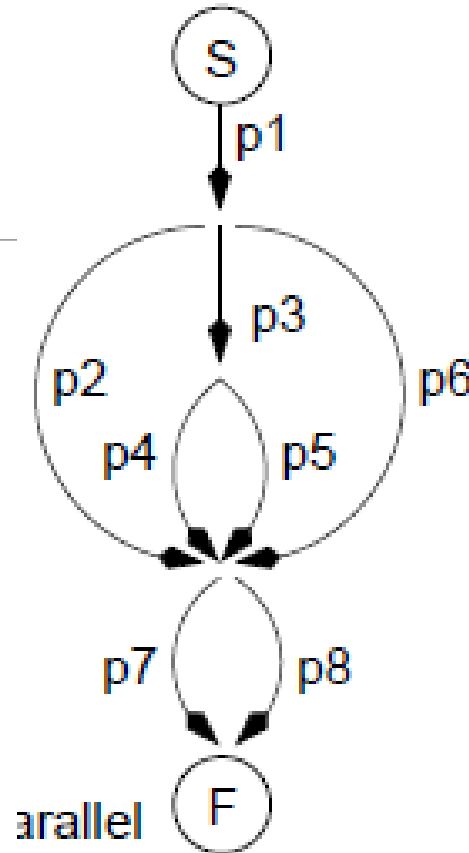
(a) Series



(b) Parallel

Graf (d) není
správně vnořený
Nelze jej popsat
kompozicí S a P

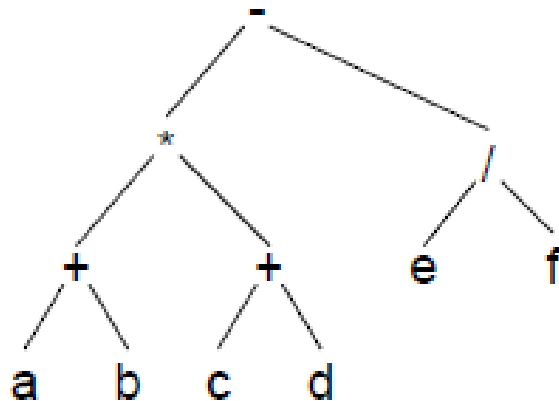
Graf vlevo lze:



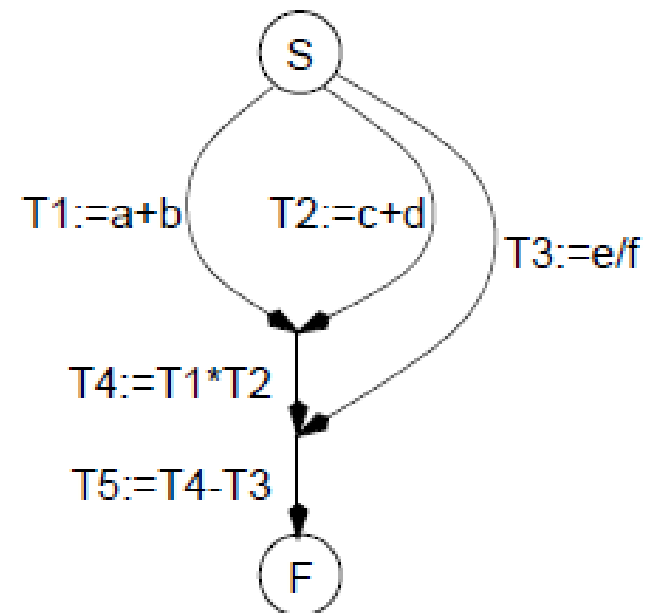
$S(p1, S(P(p2, P(S(p3, P(p4, p5)), p6)), P(p7, p8))$

Příklad vyhodnocení aritmetického výrazu

$$(a + b) * (c + d) - (e / f)$$



a) expression tree



b) process flow graph

Vznikají správně vnořené procesy; dodržet maximální paralelismus !

Abstraktní primitiva

cobegin, coend

Dijkstra (1968), původně parbegin,...

Specifikuje sekvence programu, která má být spuštěna paralelně

cobegin

$C_1 \parallel C_2 \parallel \dots \parallel C_n$

coend

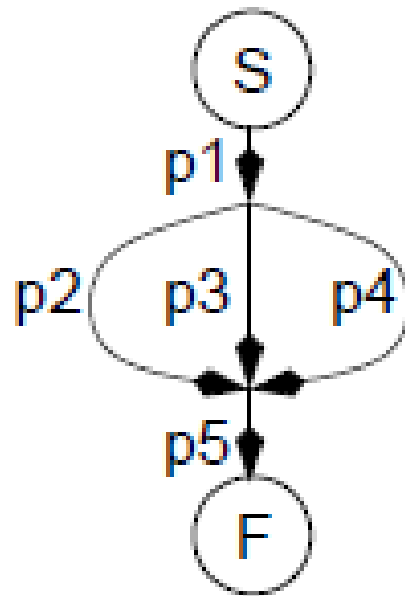
Každé C_i ... autonomní segment kódu (blok)

Samostatné vlákno pro všechna C_i

C_i běží nezávisle na ostatních

Program pokračuje za coend až po skončení posledního C_i

Příklad – cobegin, coend



```
begin
  C1;
  cobegin
    C2 || C3 || C4
  coend
  C5
end
```

Vztah cobegin/coend a funkcí P, S

Každý segment kódu C_i lze dekomponovat na sekvenci příkazů p_i :
 $S(p_{i1}, S(p_{i2}, \dots))$

Konstrukce cobegin $C_1 \parallel C_2 \parallel \dots$ coend
odpovídají vnoření funkcí:

$P(C_1, P(C_2, \dots))$

Příklad – aritmetický výraz

$$(a+b) * (c+d) - (e/f)$$

```
begin
  cobegin
    begin
      cobegin
        T1 = a+b || T2 = c+d
      coend
      T4 = T1 * T2
    end
    || T3 = e/f
  coend
  T5 = T4 - T3
end
```

**Maximální
paralelismus**

Část výpočtu
spustím ihned
jak je to možné

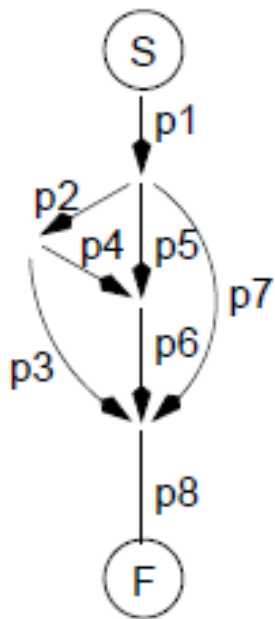
Např. T1,T2,T3

Příklad – fork, join, quit

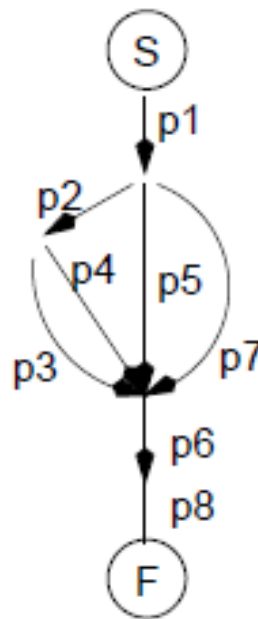
$(a+b) * (c+d) - (e/f)$

```
    n := 2;
    fork L3;
    m := 2;
    fork L2;
    t1 := a + b;    join m, L4; quit;
L2:  t2 := c + d;    join m, L4; quit;
L4:  t4 := t1 * t2;  join n, L5; quit;
L3:  t3 := e/f;      join n, L5; quit;
L5:  t5 := t4 - t3;
```

Lze nesprávně vnořený graf upravit?



(a) General



(b) "properly nested"

Můžeme „beztrestně“
posunout proces p6?

Ne vždy !!

Pokud jsou závislé, a p6
musí běžet paralelně s p3 a
p7, např. si vyměňují
zprávy, pak toto nelze.

Fork – join – quit
popíše i nesprávně
vnořené grafy

Vlákná v systému UNIX a jazyce C

- Knihovna **libpthread**
- Jako vlákno se spustí určitá funkce
- Návratem z této funkce vlákno zanikne

Základní funkce

funkce	popis
t = pthread_create(..f..)	Podprogram f se spustí jako vlákno vrací id vlákna
pthread_exit ()	Odpovídá quit, může předat návratovou hodnotu
x = pthread_join (t)	Čeká na dokončení vlákna t vrací hodnotu předanou voláním exit
pthread_detach (t)	Na dokončení vlákna se nebude čekat joinem
pthread_cancel (t)	Zruší jiné vlákno uvnitř stejného procesu

zkuste: `man pthread_create`

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

void *vlakno(void *m) /* podprogram pro vlákno */
{
    int i;

    for (i=0; i<10000; i++)
        write(1, m, 1);
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t th1, th2;

    pthread_create(&th1, NULL, vlakno, ""); /* vytvoří vlákno */
    pthread_create(&th2, NULL, vlakno, ".");
    pthread_join(th1, NULL);                /* čeká na dokončení vlákna */
    pthread_join(th2, NULL);

    return 0;
}
```

Překlad programu s vlákny

na stroji eryx.zcu.cz:

```
gcc -lpthread -o jedna jedna.c
```

```
./jedna
```

gcc .. překladač

-lpthread .. použijeme knihovnu vláken (ne vždy třeba)

-o jedna .. výsledný spustitelný soubor

jedna.c .. zdrojový kód v C

./jedna .. spustíme program z akt. adresáře