

# ZOS přednášky = struktura OS (!)

## ■ modul pro správu procesů

- program, proces, vlákno, plánování procesů a vláken
- kritická sekce, synchronizace (semaforey, ...)
- deadlock, vyhladovění

## ■ modul pro správu paměti

- virtuální paměť: stránkování, segmentace

## ■ modul pro správu I/O

## ■ modul pro správu souborů

## ■ síťování

## ■ bezpečnost

# OS reálného času (!)

- Výsledek má smysl, pouze pokud je získán v nějakém omezeném čase
- Přísné požadavky aplikací na čas odpovědi
  - Řídící počítače, multimedia
- Časově ohraničené požadavky na odpověď
  - Řízení válcovny plechu, výtahu mrakodrapu ☺
- Nejlepší snaha systému
  - Multimedia, virtuální realita
- Př: RTLinux, RTX Windows, VxWorks

# Základní funkce operačního systému (!)

- správa procesů
- správa paměti
- správa souborů
- správa zařízení - I/O subsystém
- síťování (networking)
- ochrana a bezpečnost
- uživatelské rozhraní

# Systémové volání – příklad (!)

1. Do vybraného registru (EAX) uložím číslo služby, kterou chci vyvolat
  - Je to podobné klasickému číselníku
  - Např. služba 1- vytvoření procesu, 2- otevření souboru, 3- zápis do souboru, 4- čtení ze souboru, 5- výpis řetězce na obrazovku atd.
2. Do dalších registrů uložím další potřebné parametry
  - Např. kde je jméno souboru který chci otevřít
  - Nebo kde začíná řetězec, který chci vypsát
3. Provedu instrukci, která mě přepne do režimu jádra
  - tedy INT 0x80 nebo sysenter
4. V režimu jádra se zpracovává požadovaná služba
  - Může se stát, že se aplikace zablokuje, např. čekání na klávesu
5. Návrat, uživatelský proces pokračuje dále

# Vyvolání služby systému (opakování)

- Parametry uložíme na určené místo
  - registry, zásobník
- Provedeme speciální instrukci (1)
  - vyvolá obsluhu v jádře
  - přepne do privilegovaného režimu
- OS převezme parametry, zjistí, která služba je vyvolána a provede službu
- návrat zpět
  - Přepnutí do uživatelského režimu

# Co znamená INT x?

- instrukce v assembleru pro x86 procesory, která generuje SW přerušení
- x je v rozsahu 0 až 255
- paměť od 0 do je 256 4bytových ukazatelů (celkem 1KB), obsahují adresu pro obsluhu přerušení – **vektor přerušení**
- HW interrupty jsou mapovány na dané vektory prostřednictvím programovatelného řadiče přerušení

# Druhy přerušení (!!)

## ■ Hardwarové přerušení (vnější)

- Přichází z **I/O zařízení**, např. stisknutí klávesy na klávesnici
- **Asynchronní** událost – uživatel stiskne klávesu, kdy se mu zachce
- Vyžádá si pozornost procesoru bez ohledu na právě zpracovávanou úlohu
- Doručovány prostřednictvím **řadiče přerušení** (umí stanovit **prioritu** přerušením, aj.)

## ■ Vnitřní přerušení

- Vyvolá je sám procesor
- Např. pokus o dělení nulou, **výpadek stránky paměti** (!!)

## ■ Softwarové přerušení

- Speciální strojová instrukce (např. zmiňovaný příklad **INT 0x80**)
- Je **synchronní**, vyvolané záměrně programem (chce službu OS)  
**volání služeb operačního systému z běžícího procesu (!!)**  
uživatelská úloha nemůže sama skočit do prostoru jádra OS, ale má právě k tomu softwarové přerušení

## ■ Doporučuji přečíst:

<http://cs.wikipedia.org/wiki/P%C5%99eru%C5%A1en%C3%AD>



# Kdy v OS použiji přerušení? (to samé z jiného úhlu pohledu)

## ■ **Systémové volání** (volání služby OS)

- Využiji softwarového přerušení a instrukce INT

## ■ **Výpadek stránky paměti**

- V logickém adresním prostoru procesu se odkazují na stránku, která není namapovaná do paměti RAM (rámec), ale je odložená na disku
- Dojde k přerušení – výpadek stránky
  - Běžící proces se pozastaví
  - Ošetří se přerušení – z disku se stránka natáhne do paměti (když je operační paměť plná, tak nějaký rámec vyhodíme dle nám známých algoritmů ☺)
  - Pokračuje původní proces přístupem nyní už do paměti RAM

## ■ **Obsluha HW zařízení**

- Zařízení si **žádá pozornost**
- Klávesnice: stisknuta klávesa
- Zvukovka : potřebuji poslat další data k přehrávání
- Síťová karta: došel paket



# Přijde-li přerušení... (!!)

- Přijde signalizace přerušení
- Dokončena rozpracovaná strojová instrukce
- Na zásobník **uložena adresa následující** instrukce, tj. kde jsme skončili a kde budeme chtít pokračovat **!!!!!!!**
- Z vektoru přerušení zjistí adresu podprogramu pro obsluhu přerušení
- Obsluha - rychlá
  - Na konci stejný stav procesoru (hodnoty registrů) jako na začátku
- Instrukce návratu RET, IRET
  - Vyzvedne ze zásobníku návratovou adresu a na ní pokračuje **!!!**
- Přerušená úloha (mimo zpoždění) nepozná, že proběhla obsluha přerušení

# Architektury OS

OS = jádro + systémové nástroje

Jádro se zavádí do operační paměti při startu a zůstává v činnosti po celou dobu běhu systému

Základní rozdělení:

- **Monolitické jádro** – jádro je jeden funkční celek
- **Mikrojádro** – malé jádro, oddělitelné části pracují jako samostatné procesy v user space
- **Hybridní jádro** - kombinace

# Mikrojádro (!)

- Model **klient – server**
- Většinu činností OS vykonávají **samostatné procesy mimo jádro** (servery, např. systém souborů)
- Mikrojádro
  - Poskytuje pouze nejdůležitější nízkoúrovňové funkce
    - Nízkoúrovňová správa procesů
    - Adresový prostor, komunikace mezi adresovými prostory
    - Někdy obsluha přerušení, vstupy/výstupy
  - Pouze mikrojádro běží v privilegovaném režimu
    - Méně pádů systému

# Mikrojádro

## ■ Výhody

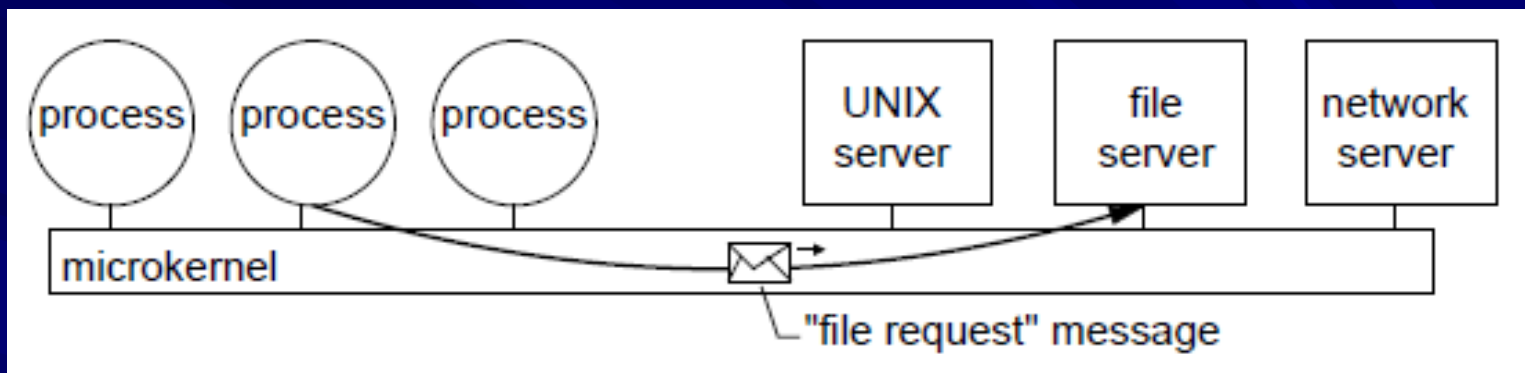
- vynucuje modulární strukturu
- Snadnější tvorba distribuovaných OS (komunikace přes síť)

## ■ Nevýhody

- Složitější návrh systému
- Režie  
(4 x přepnutí uživatelský režim  $\leftrightarrow$  jádro)

## ■ Příklady: QNX, Hurd, OSF/1, MINIX, Amoeba

# Mikrojádro



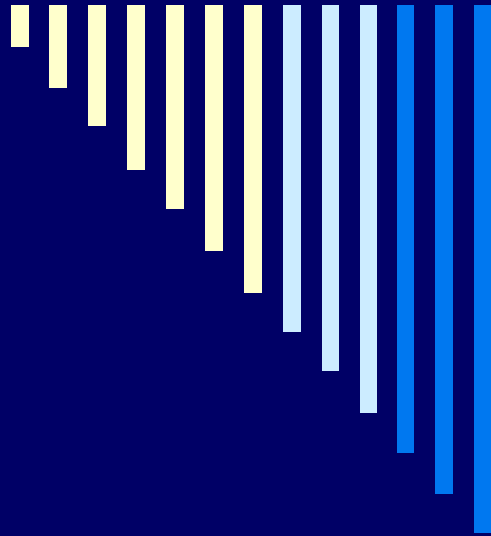
Mikrojádro – základní služby, běží v privilegovaném režimu

1. proces vyžaduje službu
2. mikrojádro předá požadavek příslušnému serveru
3. server vykoná požadavek

Snadná vyměnitelnost serveru za jiný

Chyba serveru nemusí být fatální pro celý operační systém  
(není v jádře) – nespadne celý systém

Server může event. běžet na jiném uzlu sítě (distribuov. syst.)



# 02. Koncepce OS

## Procesy, vlákna

**ZOS 2014**

---

uživatel Pepa (UID= 1015)

CPU ví, zda je v uživatelském nebo  
privilegovaném režimu

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

uživatelský  
režim

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

privilegovaný  
režim

index 128

vektor  
přerušení

0



1023

RAM

vstupní bod jádra, dle EAX  
zavolá příslušné volání

jednotlivá systémová  
volání

256 indexů  
0 .. 255  
každá položka  
obsahuje  
4B adresu  
obslužné rutiny

vektor přerušení zabírá 1KB paměti od  
adresy 0 v RAM (tzv. reálný režim CPU)





# Co se děje při obsluze přerušení?

1. na zásobník uložíme **návratovou adresu (CS:IP)**, kde budeme dále pokračovat, přepnutí do **privilegovaného** režimu
2. na zásobník uložíme hodnoty registrů
3. ... provede se obsluha ...
4. ze zásobníku vybereme hodnoty registrů ze zásobníku vybereme **návratovou adresu (CS:IP)** určující instrukci, kde bude náš proces pokračovat a přepnutí do **uživatelského** režimu



# Procesy

- **Proces** – instance běžícího programu
- **Adresní prostor** procesu
  - MMU (Memory Management Unit) zajišťuje soukromí
  - 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

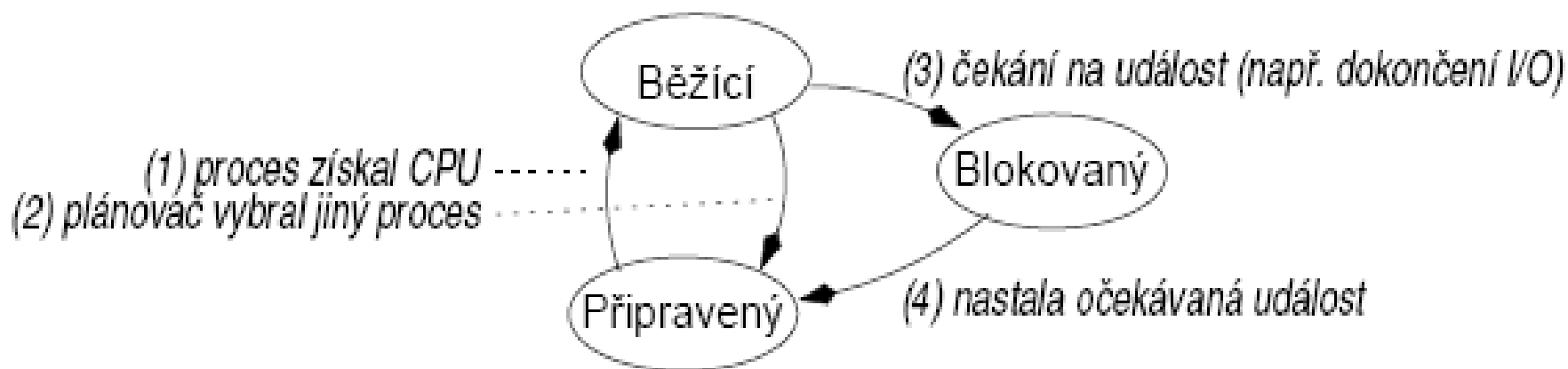


# Rychlost procesů

- ❑ Rychlost běhu procesu **není konstantní**.
- ❑ Obvykle **není** ani **reprodukovatelná**.
- ❑ Procesy nesmějí mít vestavěné **předpoklady o časování (!)**
- ❑ Např. doba trvání I/O různá.
- ❑ 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..

# Základní stavy procesu (!!)





# 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. jeho běh 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 instrukci - PC
    - ukazatel zásobníku SP
    - Stav CPU – PSW (Program Status Word)
  - 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
-



# Položky – správa souborů

## □ Nastavení prostředí

- Aktuální pracovní adresář

## □ Otevřené soubory

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



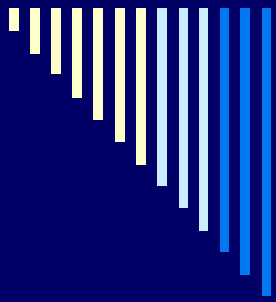
# PCB

Pointer	Process state
Process number	
Program counter	
<b>Registers</b>	
Memory limits	
List of open files	
...	



# Ukončení procesu - možnosti

- I. Proces úspěšně vykoná kód programu 😊
  - II. Skončí rodičovský proces a OS nedovolí pokračovat child procesu (záleží na OS, někdy ano někdy ne)
  - III. Proces překročí limit nějakého zdroje
-



# 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)



# Vlákna (!!)

- Vlákna v procesu **sdílejí** adresní prostor, otevřené soubory (atributy procesu)
- Vlákna **mají soukromý** čítač instrukcí, obsah registrů, soukromý zásobník
  - Mohou mít soukromé lokální proměnné
- Původně využívána zejména pro VT výpočty na multiprocесorech (každé vlákno vlastní CPU, společná data)



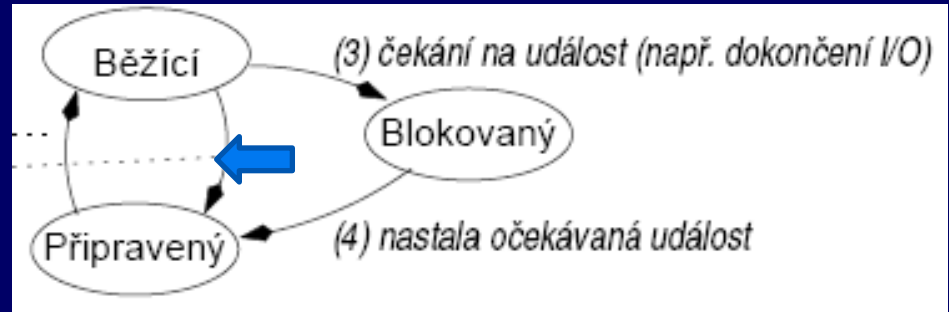
# Plánování

## □ Npreemptivní

- Proces skončí
- Běžící -> Blokováný
  - Čekání na I/O operaci
  - Čekání na semafor
  - Čekání na ukončení potomka

## □ Preemptivní

- Navíc přechod:  
Běžící -> Připravený
  - Uplynulo časové kvantum



Proces opustí CPU:  
jen když skončí,  
nebo se zablokuje

Preemptivní  
navíc opustí CPU při uplynutí  
časového kvanta  
Problém – proces může být  
přerušen kdykoliv, bohužel i v  
nevhodný čas



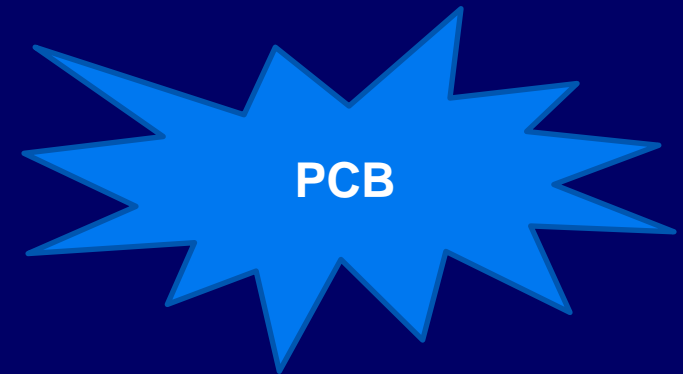
# základní funkce (!!)

funkce	popis
<b>pthread_create</b>	Vytvoří nové vlákno. Jako kód vlákna se bude vykonávat funkce, která je zadaná jako parametr této funkce Defaultně je vytvořené vlákno v joinable stavu.
<b>pthread_join</b>	Čeká na dokončení jiného vlákna, vlákno na které se čeká musí být v joinable stavu
<b>pthread_detach</b>	Vlákno bude v detached stavu – nepůjde na něj čekat pomocí pthread_join Paměťové zdroje budou uvolněny hned, jak vlákno skončí (x zabrání synchronizaci)
<b>pthread_exit</b>	Naše vlákno končí, když doběhne funkce, kterou vykonává, nebo když zavolá pthread_exit



# Proces UNIXU – obsahuje informace:

- ❑ Proces ID, proces group ID, user ID, group ID
- ❑ Prostředí
- ❑ Pracovní adresář
- ❑ Instrukce programu
- ❑ Registry
- ❑ Zásobník (stack)
- ❑ Halda (heap)
- ❑ Popisovače souborů (file descriptors)
- ❑ Signal actions
- ❑ Shared libraries
- ❑ IPC (fronty zpráv, roury, semaforey, sdílená paměť)





# Vlákno má vlastní (!!):

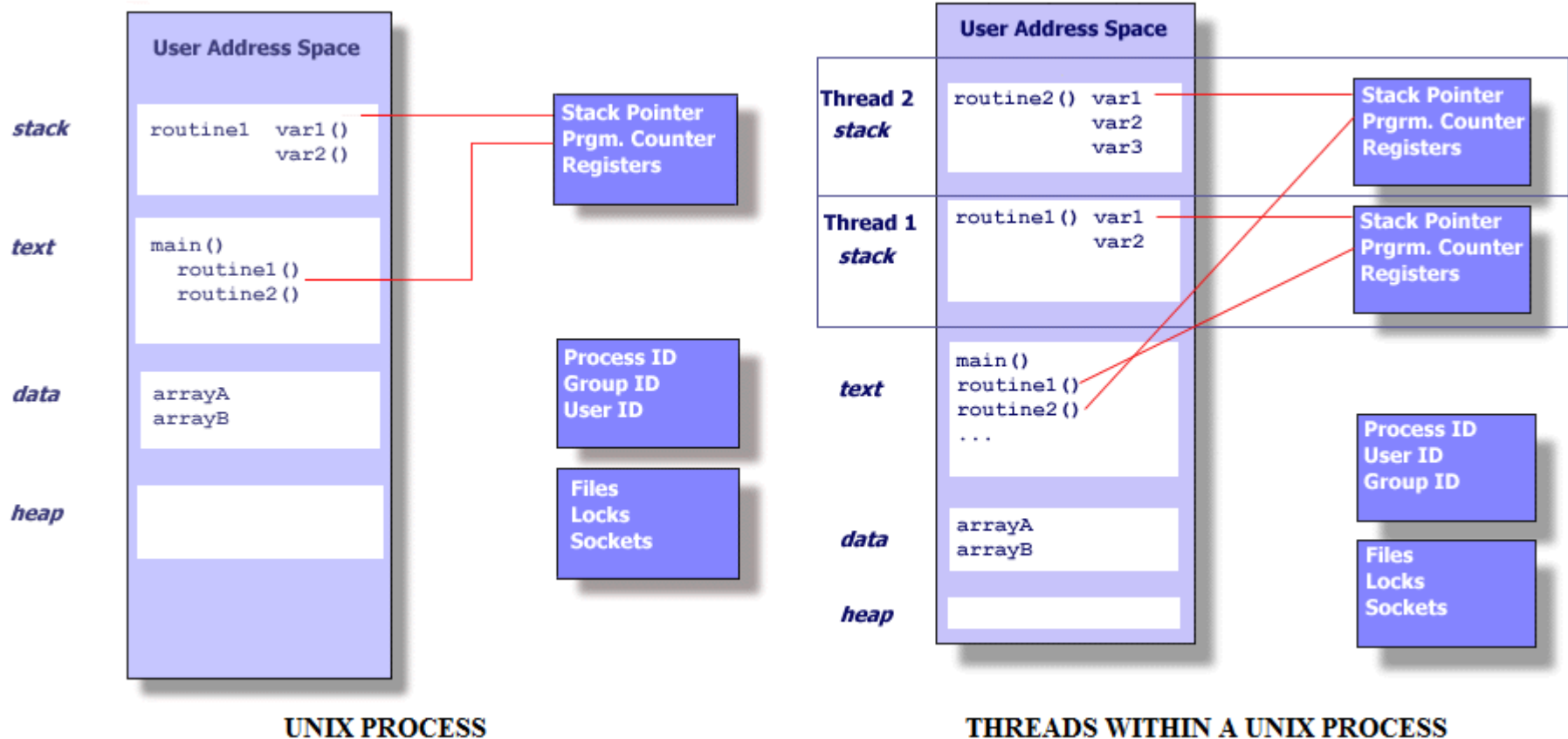
- ❑ Zásobník (stack pointer)
- ❑ Registry
- ❑ Plánovací vlastnosti (policy, priority)
- ❑ Množina pending a blokováných signálů
- ❑ Data specifická pro vlákno

Všechna vlákna uvnitř stejného procesu sdílejí stejný adresní prostor

Mezivláknová komunikace je efektivnější a snadnější než meziprocsová

---

proces vs. proces s více vlákn  
(rozdělení paměti je jen ilustrativní)



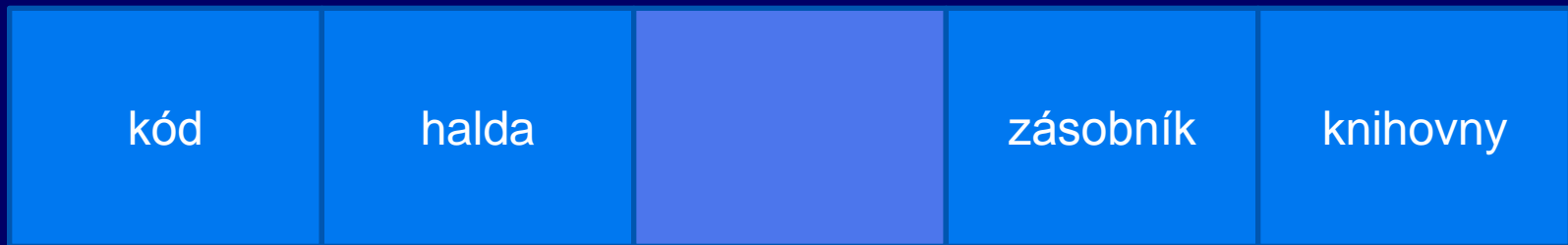


# Rozdělení paměti pro proces

Roste halda



Roste zásobník



Máme-li více vláken => více zásobníků, limit velikosti zásobníku



Vytvořené vlákno





# Výskyt souběhu

- časový souběh se projevuje **nedeterministicky**
  - **může nastat kdykoliv**
- většinu času běží programy bez problémů
- hledání chyby je obtížné





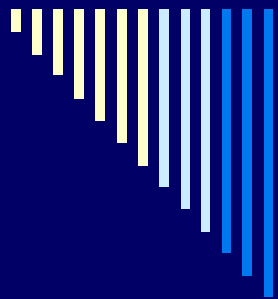
# Řešení časového souběhu

- pokud **čtení a modifikace atomicky**
  - atomicky = jedna nedělitelná operace
  - souběh nenastane
- **hw** většinou není praktické zařídit
- **sw řešení**
  - v 1 okamžiku dovolíme číst a zapisovat společná data **pouze 1mu procesu**
  - => ostatním procesům **zabránit**



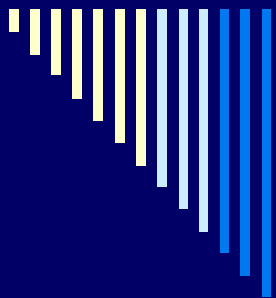
# Kritická sekce

- sekvenční procesy
    - komunikace přes společnou datovou oblast
  - **kritická sekce** (critical section, region)
    - místo v programu, kde je prováděn přístup ke společným datům
  - úloha – jak implementovat, aby byl v kritické sekci v daný okamžik pouze 1 proces
-



# Pravidla pro řešení časového souběhu (!)

1. **Vzájemné vyloučení** - žádné dva procesy nesmějí být **současně** uvnitř své kritické sekce
2. Proces běžící **mimo kritickou sekci nesmí blokovat** jiné procesy (např. jim **bránit ve vstupu** do kritické sekce)
3. Žádný proces nesmí na vstup do své kritické sekce **čekat nekonečně dlouho** (jiný vstupuje **opakovaně**, neumí se **dohodnout** v konečném čase, kdo vstoupí první)



# Možnosti řešení

- Zákaz přerušení
- Aktivní čekání
- Zablokování procesu



# Zákaz přerušení

- vadí nám přeplánování procesů
  - **výsledek přerušení** v systémech se sdílením času
- zákaz přerušení → k přepínání nedochází
  - **zakaž** přerušení;
  - **kritická sekce**;
  - **povol** přerušení;



# Zákaz přerušení II.

- nejjednodušší řešení – na uniprocessoru (1 CPU)
  - není dovoleno v **uživatelském režimu** (jinak by uživatel zakázal přerušení a už třeba nepovolil...)
  - používáno často **uvnitř jádra** OS
  - ale **není** vhodné pro **uživatelské procesy**
-



# Aktivní čekání

## □ Aktivní čekání

- **Průběžné testování** proměnné ve smyčce, dokud nenabude očekávanou hodnotu

## □ Většinou se snažíme **vyhnout**

- **plýtvá** časem CPU

## □ Používá se, pokud předpokládáme **krátké čekání**

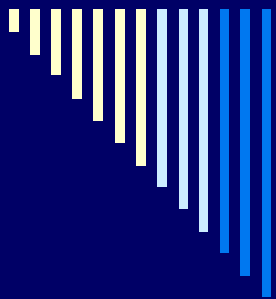
- **spin lock**
-



# Spin lock s instrukcí TSL (!!)

- hw podpora:
- většina počítačů – instrukci, která **otestuje hodnotu a nastaví paměťové místo v jedné nedělitelné operaci**
  
- operace **Test and Set Lock** – TSL, TS:
- **TSL R, lock**
  - LD R, lock
  - LD lock, 1
  
- R je registr CPU
- lock – buňka paměti, 0 false nebo 1 true; boolean;





# TSL

- Provádí se nedělitelně (atomicky) – žádný proces nemůže k proměnné lock přistoupit do skončení TSL
- Multiprocessor – zamkne paměťovou sběrnici po dobu provádění instrukce



# Struktura semaforu (!!)

```
typedef struct {  
    int hodnota;  
    process_queue *fronta;  
} semaphore;
```

hodnota semaforu: 0, 1, 2, ..

fronta procesů čekajících na  
daný semafor



# Operace P(!!)

## Operace P(S):

if  $S > 0$

$S--$ ;

else

zablokuj\_proces;

zablokuje proces, který chtěl provést operaci P:

- přidá jej do fronty procesů čekajících na daný semafor
- stav procesu označí jako blokový



# Operace V(!!)

## Operace V(S):

```
if (proces_blokovany_nad_semaforem)
    jeden_proces_vzbud;
else
    S++;
```

podívá se, zda je  
fronta prázdná či ne

označí stav procesu  
jako připravený  
vyjme proces z fronty  
na semafor

(Pokud je nad semaforem S **zablokovaný** jeden nebo více procesů,  
**vzbudí** jeden z procesů; proces pro vzbuzení je vybrán **náhodně**)

# Pamatuj

Semafor je tvořen celočíselnou proměnnou **s** a frontou procesů, které čekají na semafor, a jsou nad ním implementovány operace **P()** a **V()**

**s** může nabývat hodnot **0, 1, 2, ..**

Hodnota 0 znamená, že je semafor zablokován, a prvolání, operace P() se daný proces zablokuje

Nenulová hodnota s znamená, kolik procesů může zavolat operaci P(), aniž by došlo k jejich zablokování

Pro **vzájemné vyloučení** je tedy počáteční hodnotu **s** potřeba nastavit na **1**, aby operaci P() bez zablokování mohl vykonat jeden proces



# Vzájemné vyloučení (!!!)

```
var s: semaphore = 1;  
cobegin  
  while true do  
    begin  
      ...  
      P(s);  
      KS1;  
      V(s);  
      ...  
    end  
  || {totéž dělá další proces}  
coend
```

Na začátku je vstup do kritické sekce volný, tedy hodnota semaforu 1

Z funkce P(s) se vrátíme, až když je vstup do kritické sekce volný

Zavoláním V(s) signalizujeme, že je kritická sekce nyní volná a dovnitř může někdo další



# Producent – konzument (!!!)

Producent – konzument je jedna ze základních synchronizačních úloh z teorie OS.

Cílem je správně synchronizovat přístup k sdílenému bufferu omezené velikosti – ošetřit mezní stavy, kdy je prázdný a naopak plný.

Měli byste umět v obecné podobě tuto úlohu vyřešit s využitím tří semaforů.



# P&K – implementace II. (!)

```
cobegin
```

```
  while true do { producent
```

```
    begin
```

```
      produkuj záznam;
```

```
      P(e);           // je volná položka?
```

```
      P(m); vlož do bufferu; V(m);
```

```
      V(f);           // zvětší obsazených
```

```
    end {while}
```

Není-li volná položka v  
bufferu, zablokuje se





# P&K – implementace III.

```
||  
while true do { konzument }  
begin  
    P(f); // je plná nějaká položka?  
    P(m); vyber z bufferu; V(m);  
    V(e); // zvětši počet prázdných  
    zpracuj záznam;  
end {while}  
coend.
```

Pokud je buffer prázdný,  
zablokuje se



# Mutexy

- Potřebujeme zajistit **vzájemné vyloučení**
  - chceme „**spin-lock**“ bez aktivního čekání
  - nepotřebujeme obecně schopnost semaforů čítat
- **mutex** – mutual exclusion
  - paměťový zámek

Mutex řeší vzájemné vyloučení a je k systému šetrnější než čistě aktivní čekání spin-lock, můžeme jej naimplementovat např. pomocí **TSL** instrukce a volání **yield**



# Implementace mutexu (!!)

## – s podporou jádra OS




Oblíbená  
instrukce  
TSL

**mutex\_lock:**TSL R, mutex

CMP R, 0

JE ok

**CALL** yield



Vzdát  
se CPU

JMP mutex\_lock

ok: RET

;; R:=mutex a mutex:=1

;; byla v mutex hodnota 0?

;; pokud byla skok na ok

;; vzdáme se procesoru -  
naplánuje se jiné vlákno

;; zkusíme znovu, později

**mutex\_unlock:**

LD mutex, 0

RET

;; ulož 0 do mutex



# Mutex x binární semafor

- Společné – použití pro vzájemné vyloučení
- Často se v literatuře mezi nimi příliš nerozlišuje
- Někdy jsou zdůrazněny rozdíly

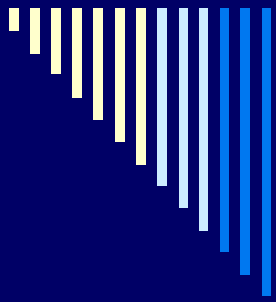
## Mutex

s koncepcí vlastnictví:

Odemknout mutex může jen stejné vlákno/proces, který jej zamkl (!!!!)

uvědomte si,  
kdy nám toto  
může vadit

pamatovat si co znamená  
pojem mutex s koncepcí  
vlastnictví



# Chyby – přehození P a V

Přehození P a V operací nad mutexem:

1. V()
2. kritická sekce
3. P()

Důsledek – více procesů může vykonávat kritickou sekci současně



# Chyby – dvě operace P

1. P()
2. Kritická sekce
3. P()

Důsledek - deadlock



# Chyby – vynechání P, V

- Proces vynechá P()
- Proces vynechá V()
- Vynechá obě

Důsledek – porušení vzájemného vyloučení nebo deadlock



# Monitor

- V monitoru může být **v jednu chvíli** **AKTIVNÍ pouze jeden** proces !!
- Ostatní procesy jsou při pokusu o vstup do monitoru pozastaveny





# Operace nad podmínkami (!!)

- Definovány 2 operace – **wait** a **signal**

## **C.wait**

- Volající bude **pozastaven nad podmínkou C**
- Pokud je některý proces připraven vstoupit do monitoru, bude mu to dovoleno

často také najdeme ve tvaru: wait(c), signal(c)



# Problém s operací **signal**

- Pokud by signál pouze vzbudil proces, běžely by v monitoru dva
    - Vzbuzený proces
    - A proces co zavolal signal
  - **ROZPOR** s definicí monitoru
    - V monitoru může být v jednu chvíli **aktivní** pouze jeden proces
  - Několik řešení
-



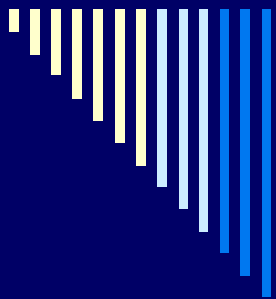
# Řešení reakce na signal (!!)

## □ Hoare

- proces volající **c.signal** se **pozastaví**
- Vzbudí se až poté co předchozí rektivovaný proces opustí monitor nebo se pozastaví

## □ Hansen

- Signal smí být uveden pouze jako **poslední** příkaz v monitoru
  - Po volání signal musí proces **opustit** monitor
-



# Monitory - Java

Jde o třetí řešení problému, jak ošetřit volání signal (k Hoare, Hansen):

Čekající může běžet až poté, co proces (vlákno) volající signál opustí monitor



---

příkaz strace

– jaká systémová volání proces volá (!!)

**strace ls je.txt neni.txt**

- bude nás zajímat program ls
  - vidíme volání `execve("/bin/ls", ...)`
  - vidíme chybový výstup `write(2, ...)`
  - vidím stand. výstup `write(1, ...)`
-



# Meziprocesová komunikace

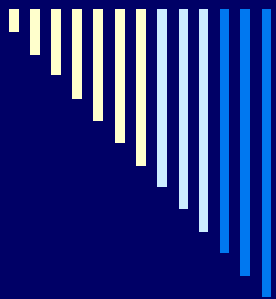
- Předávání zpráv
  - Primitiva send, receive
  - Mailbox, port
  - RPC
  - Ekvivalence semaforů, zpráv, ...
  - Bariéra, problém večeřících filozofů
-



# Meziprocesová komunikace

Procesy mohou komunikovat:

- Přes **sdílenou paměť**  
(předpoklad: procesy na stejném uzlu)
- **Zasíláním zpráv**  
(na stejném uzlu i na různých uzlech)



# Synchronizace

- **blokující** (synchronní)
- **neblokující** (asynchronní)
  - čeká **send** na převzetí zprávy příjemcem?
  - co když při **receive** není žádná zpráva?
- většinou **send neblokující**, **receive blokující (!)**





# Mailbox, port

Termíny používané v teorii OS, neplést s pojmem mailbox jak jej běžně znáte

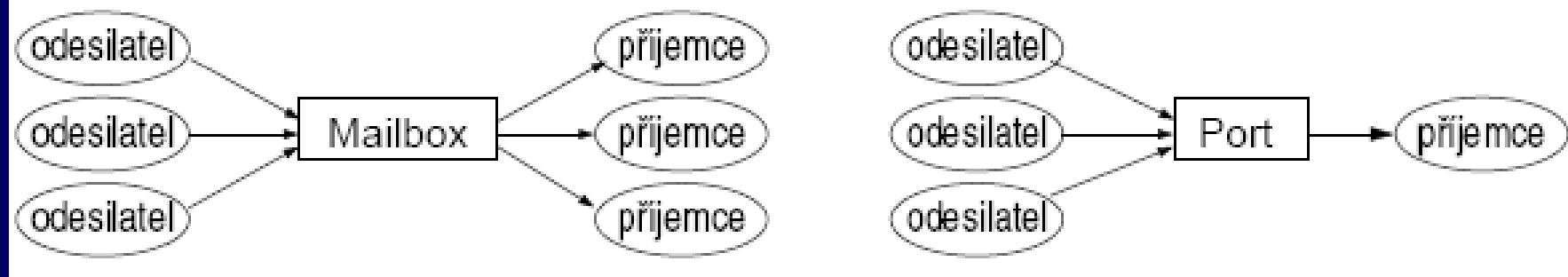
## □ mailbox

- fronta zpráv využívaná **více** odesílateli a příjemci
- obecné schéma
- operace receive – drahá, zvláště pokud procesy běží na různých strojích

## □ port

- omezená forma mailboxu
  - zprávy může vybírat **pouze jeden** příjemce
-

# Mailbox, port





# Lokální komunikace (!)

Na stejném stroji – snížení režie na zprávy

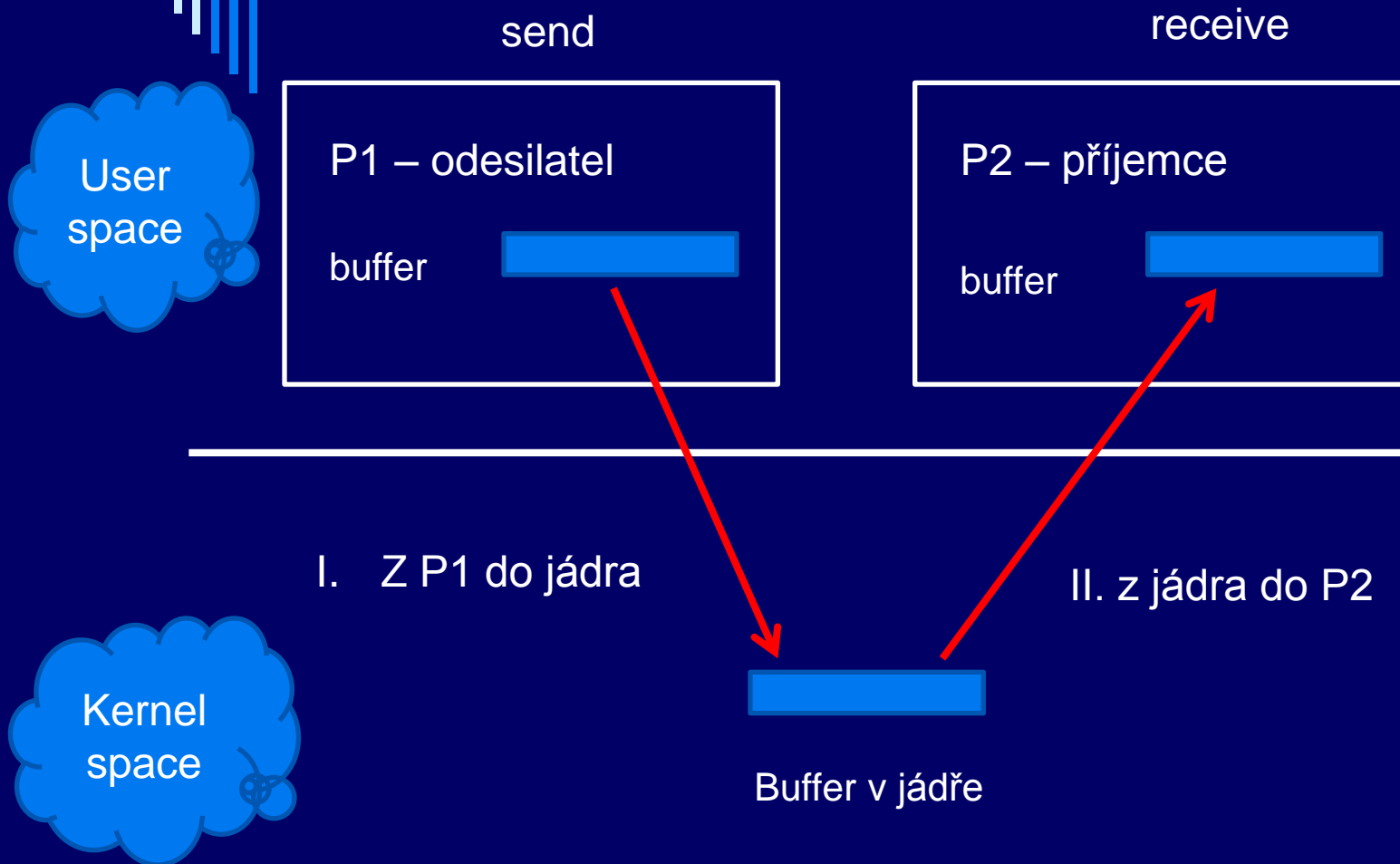
## □ Dvojí kopírování (!)

- z procesu odesílatele do fronty v jádře
- z jádra do procesu příjemce

## □ rendezvous

- **eliminuje frontu zpráv**
- např. send zavolán dříve než receive – odesílatel zablokován
- až vyvoláno obojí, send i receive – zprávu zkopírovat z odesílatele **přímo** do příjemce
- efektivnější, ale méně obecné (např. jazyk ADA)

# Dvojí kopírování (!!!)





# Lokální komunikace II.

dvojí kopírování a rendezvous – to co se používá  
zde úvahy, zda by šlo dále zefektivnit

- využití mechanismu **virtuální paměti (!)**
  - další možnost, lze využít např. při rendezvous
  - paměť obsahující zprávu je **přemapována**
    - z procesu odesílatele
    - do procesu příjemce
  - zpráva se **nekopíruje**
    - Virtuální paměť umí “čarovat” komu daný kus fyzické paměti namapuje a na jaké adresy

# opakování (!!)

v Linuxu je datová struktura `task_struct`, která obsahuje informace o procesu (tj. představuje PCB)

- každý proces má záznam (řádku) v **tabulce procesů**
- tomuto záznamu se říká **PCB** (process control block)
- PCB obsahuje všechny potřebné informace (tzv. **kontext** procesu) k tomu, abychom mohli proces **kdykoliv pozastavit** (odejmout mu procesor) a znovu jej od tohoto místa přerušení spustit (Program Counter: CS:EIP)
- proces po opětovném přidělení CPU pokračuje ve své činnosti, jako by k žádnému přerušení vykonávání jeho kódu nedošlo, je to z jeho pohledu transparentní

# opakování (!!)

- kde leží tabulka procesů?  
v paměti RAM, je to datová struktura jádra OS
- kde leží informace o PIDu procesu?  
v tabulce procesů -> v PCB (řádce tabulky) tohoto procesu
- jak procesor ví, kterou instrukci procesu (vlákna) má vykonávat?  
podle program counteru (PC, typicky CS:EIP), ukazuje na oblast v paměti, kde leží vykonávaná instrukce;  
obsah CS:EIP, stejně jako dalších registrů je součástí PCB

# Pamatuj

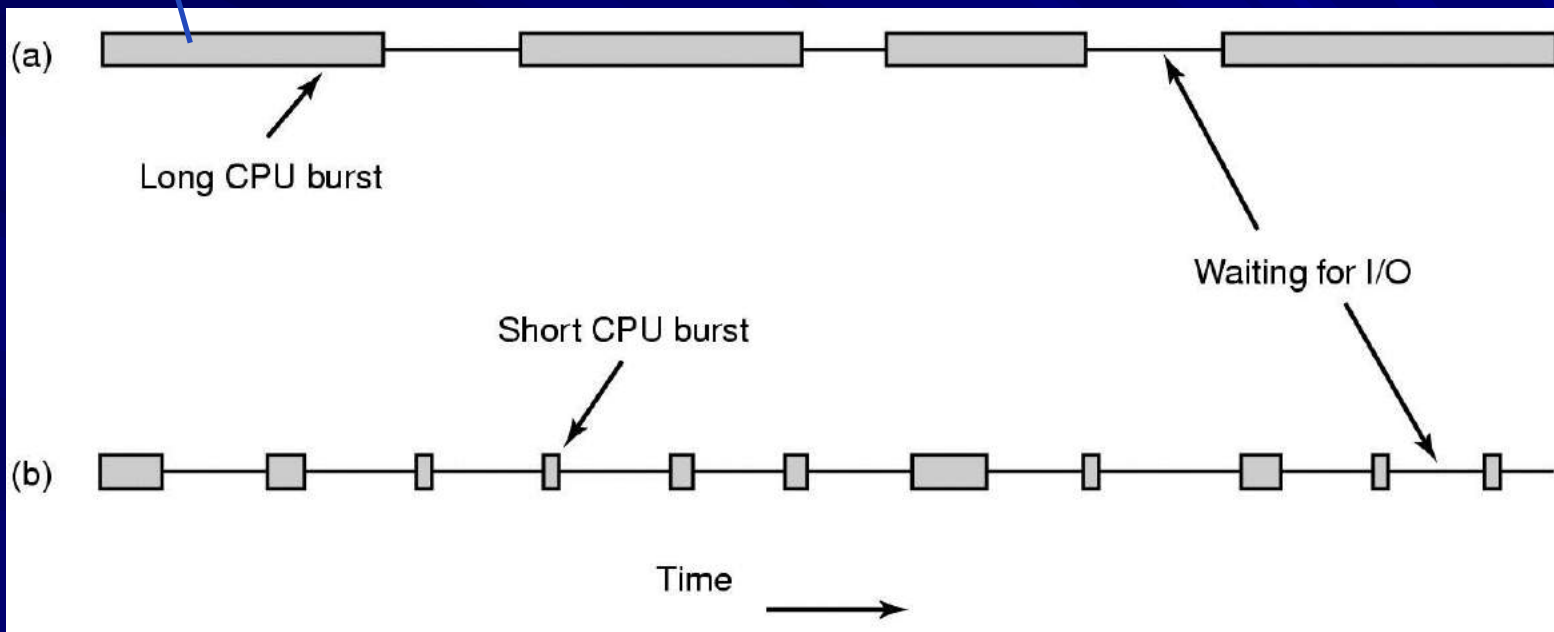
**Plánovač** určí, který proces (vlákno) by měl běžet nyní.

**Dispatcher** provede vlastní přepnutí z aktuálního běžícího procesu na nově vybraný proces.



počítám

# Plánování



- a) CPU-vázaný proces („hodně času tráví výpočtem“)
- b) I/O vázaný proces („hodně času tráví čekáním na I/O“)

Uveďte příklady CPU vázaného a I/O vázaného procesu

# Preemptivní vs. non-preemptivní plánování

## ■ Non-preemptivní

- každý proces dokončí svůj CPU burst (!!!!)
- proces si podrží kontrolu nad CPU, dokud se jí nevzdá (I/O čekání, ukončení)
- lze v dávkových systémech, není příliš vhodné pro time sharingové (se sdílením času)
- Win 3.x non-preemptivní (kooperativní) plánování
- od Win95 preemptivní
- od Mac OS 8 preemptivní
- na některých platformách je stále

Jaký má vliv non-preemptivnost systému na obsluhu kritické sekce u jednojádrového CPU?

# Preemptivní vs. non-preemptivní plánování

## ■ Preemptivní plánování

- proces lze přerušit **KDYKOLIV** během CPU burstu a naplánovat jiný (-> problém kritických sekcí !!!)
- dražší implementace kvůli přepínání procesů (režie)
- Vyžaduje speciální hardware – **timer (časovač)**  
časovač je na základní desce počítače, pravidelně generuje hardwarová přerušení systému od časovače

Část výkonu systému spotřebuje režie nutná na přepínání procesů. K přepnutí na jiný proces také může dojít v nevhodný čas (ošetření KS).

Preemptivnost je ale u současných systémů důležitá, pokud potřebujeme interaktivní odezvu systému.

Časovač tiká (generuje přerušení), a po určitém množství tiků se určí, zda procesu nevypršelo jeho časové kvantum.

# Plánovač (!)

## ■ rozhodovací mód

- okamžik, kdy jsou vyhodnoceny priority procesu a vybrán proces pro běh

## ■ prioritní funkce

- určí prioritu procesu v systému

## ■ rozhodovací pravidla

- jak rozhodnout při stejné prioritě

Tři zásadní údaje, které charakterizují plánovač

# Prioritní funkce (!)

priorita = statická + dynamická

proč 2 složky?

pokud by chyběla:

- statická – nemohl by uživatel např. při startu označit proces jako důležitější než jiný
- dynamická – proces by mohl vyhladovět, mohl by být neustále předbíhán v plánování jinými procesy s větší prioritou

# Plánovač – Prioritní funkce

Co všechno může vzít v úvahu prioritní funkce:

- čas, jak dlouho proces využíval CPU
- aktuální zatížení systému
- paměťové požadavky procesu
- čas, který strávil v systému
- celková doba provádění úlohy (limit)
- urgency (RT systémy)

# Plánovač – Rozhodovací pravidlo

- malá pravděpodobnost stejné priority
  - náhodný výběr
- velká pravděpodobnost stejné priority
  - cyklické přidělování kvanta
  - chronologický výběr (FIFO)

Prioritní funkce může být navržena tak, že málokdy vygeneruje stejné priority, nebo naopak může být taková, že často (nebo když se nepoužívá vždy) určí stejnou hodnotu. Pak nastupuje rozhodovací pravidlo.

# Dávkové systémy (!)

## ■ průchodnost (throughput)

- počet úloh dokončených za časovou jednotku

## ■ průměrná doba obrátky (turnaround time)

- průměrná doba od zadání úlohy do systému do dokončení úlohy

## ■ využití CPU

Průchodnost a průměrná doba obrátky jsou různé údaje ! Někdy snaha vylepšit jednu hodnotu může zhoršit druhou z nich.



# Plánování úloh v dávkových systémech

- FCFS (First Come First Served)
- SJF (Shortest Job First)
- SRT (Shortest Remaining Time)
  - Preemptivní varianta SJF
- Multilevel Feedback

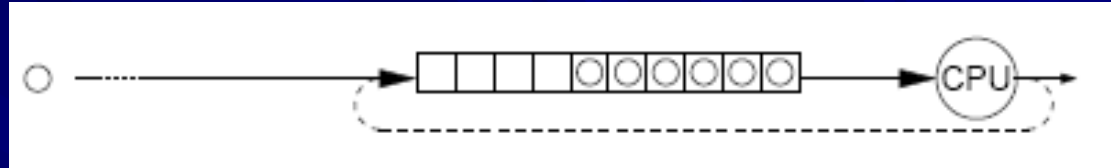
# FCFS (First Come First Served)

■ FIFO

■ Nepreemptivní FIFO

■ Základní varianta

- Nově příchozí na konec fronty připravených
- Úloha běží dokud neskončí, poté vybrána další ve frontě (viz 1.)



■ Co když úloha provádí I/O operaci?

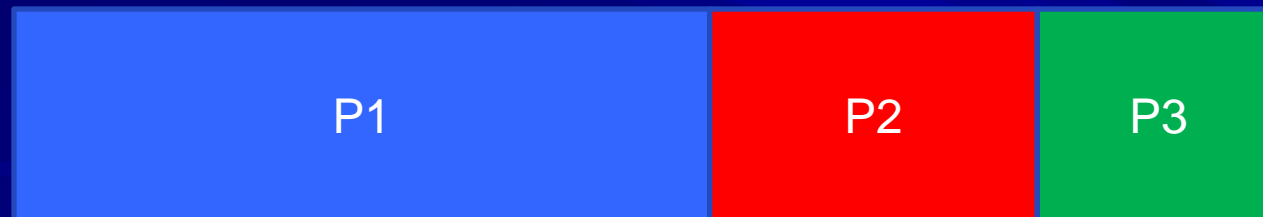
1. Zablokována, CPU se nevyužívá (základní varianta)
  2. Do stavu blokováný, běží jiná úloha po dokončení I/O zařazena na **konec** fronty připravených (častá varianta !!!)
- I/O vázané úlohy znevýhodněny před výpočetně vázanými
3. Další možná modifikace → po dokončení I/O na začátek fronty připravených

# FCFS příklad

V čase nula budou v systému procesy P1, P2, P3 přišlé v tomto pořadí.

proces	Doba trvání (s)
P1	15
P2	5
P3	4

doba obrátky:  
odešel  
-  
přišel



0

15

20

24

tedy:

P1:  $15 - 0 = 15$

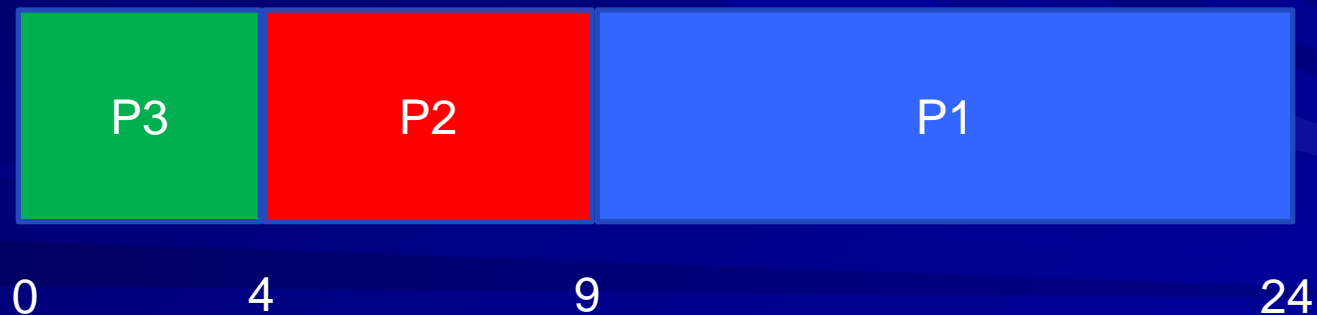
P2:  $20 - 0 = 20$

P3:  $24 - 0 = 24$

průměrná doba obrátky:  $(15+20+24) / 3 = 19,666$

# SJF (Shortest Job First)

- Nejkratší úloha jako první
- Předpoklad – známe přibližně dobu trvání úloh
- **Nepreemptivní**
  - Jedna fronta příchozích úloh
  - Plánovač vybere vždy úlohu s nejkratší dobou běhu
- Optimalizuje dobu obrátky



průměrná doba obrátky:  $(4+9+24) / 3 = 12,3$

# SRT (Shortest Remaining Time)

- Úlohy můžou přicházet **kdykoliv** (*nejen v čase nula*)
- **Preemptivní** (!! možný přechod běžící - připravený)
  - Plánovač vždy vybere úlohu, jejíž **zbývající** doba běhu je nejkratší (!!!)
- Př. KDY dojde k preempci:  
Právě prováděné úloze zbývá 10 minut, do systému právě teď přijde úloha s dobou běhu 1 minutu – systém **prováděnou** úlohu **pozastaví** a nechá běžet novou úlohu
  - i když by byla tvořena jen CPU burstem
- Možnost vyhladovění dlouhých úloh (!) => neustále předbíhány krátkými

# Multilevel feedback

- **N** prioritních úrovní
- Každá úroveň má svojí frontu úloh
- Úloha vstoupí do systému do fronty s **nejvyšší** prioritou (!)
- Na každé prioritní úrovni
  - Stanoveno maximum času CPU, který může úloha obdržet
  - Např.:  $T$  na úrovni  $n$ ,  $2T$  na úrovni  $n-1$  atd.
  - Pokud úloha překročí tento limit, její priorita se sníží
  - Na nejnižší prioritní úrovni může úloha běžet neustále nebo lze překročení určitého času považovat za chybu
- Proces obsluhuje nejvyšší neprázdnou frontu (!!)

# Shrnutí – dávkové systémy

algoritmus	Rozh. mód	Prioritní funkce	Rozh. pravidlo
FCFS	Nepreemptivní	$P(r) = r$	Náhodně
SJF	Nepreemptivní	$P(t) = -t$	Náhodně
SRT	Preemptivní (při příchodu úlohy)	$P(a,t) = a-t$	FIFO nebo náhodně
MLF	nepreemptivní	Viz popis ☺	FIFO v rámci fronty

r celkový čas strávený úlohou v systému  
t předpokládaná délka běhu úlohy  
a čas strávený během úlohy v systému



# Algoritmus cyklické obsluhy – Round Robin (RR)

- jeden z nejstarších a nejpoužívanějších
- každému procesu přiřazen **časový interval**
  - **časové kvantum**, po které může běžet
- proces běží na konci kvanta
  - **preemce**, naplánován a spuštěn další připravený proces
- proces skončí nebo se zablokuje před uplynutím kvanta
  - **stejná akce** jako v předchozím bodě ☺





# Dynamická priorita

- V kvantově orientovaných plánovacích algoritmech:
- dynamická priorita např. dle vzorce  $1 / f$  (!)
- $f$  – velikost části kvanta, kterou proces naposledy použil
- zvýhodní I/O vázané x CPU vázaným

Pokud proces nevyužil celé kvantum, jeho dynamická priorita se zvyšuje, např. pokud využil posledně jen 0.5 kvanta, tak  $1/0,5 = 2$ , pokud celé kvantum využil  $1/1=1$



# Spojení cyklického a prioritního plánování

- **prioritní třídy**
  - v každé třídě procesy se stejnou prioritou
- **prioritní plánování** mezi třídami
  - Bude obsluhována třída s nejvyšší prioritou
- **cyklická obsluha** uvnitř třídy
  - V rámci dané třídy se procesy cyklicky střídají
- obsluhovány jsou pouze připravené procesy v nejvyšší neprázdné prioritní třídě

A kdy se dostane na další fronty?



# Plánovač spravedlivého sdílení

## □ problém:

- čas přidělován každému procesu nezávisle
- Pokud uživatel má více procesů než jiný uživatel  
-> dostane více času celkově

## □ spravedlivé sdílení

- přidělovat čas každému **uživateli** (či jinak definované skupině procesů) **proporcionálně**, bez ohledu na to, kolik má procesů
- máme-li **N uživatelů**, každý dostane  **$1/N$  času**

= spravedlnost vůči uživatelům



# Plánování pomocí loterie

- Lottery Scheduling (Waldspurger & Weihl, 1994)
- cílem – poskytnout procesům příslušnou proporci času CPU
- základní princip:
  - procesy obdrží **tikety (losy)**
  - plánovač **vybere náhodně** jeden tiket
  - **vítězný** proces obdrží cenu – 1 **kvantum** času CPU
  - důležitější procesy – **více tiketů**, aby se zvýšila šance na výhru (celkem 100 losů, proces má 20 – v dlouhodobém průměru dostane 20% času)



# Loterie - výhody

řešení problémů, v jiných plán. algoritmech obtížné

- **spolupracující procesy – mohou si předávat losy**
  - klient posílá zprávu serveru a blokuje se
  - může serveru **propůjčit** všechny **své tikety**
  - po vykonání požadavku server tikety **vrátí**
  - nejsou-li požadavky, server žádné tikety nepotřebuje

# Shrnutí

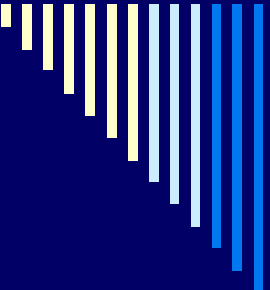
Algoritmus	Rozhodovací mód	Prioritní funkce	Rozhodovací pravidlo
RR	Preemptivní vyprší kvantum	$P() = 1$	cyklicky
prioritní	Preemptivní $P \text{ jiný} > P$	Viz text	Náhodně, cyklicky
spravedlivé	Preemptivní $P \text{ jiný} > P$	$P(p,g)=p-g$	cyklicky
loterie	Preemptivní vyprš. kv.	$P() = 1$	Dle výsledku loterie

# afinita

na jakých CPU může daný proces běžet

správce úloh systému  
Windows – procesy –  
vybrat proces – pravá myš  
– nastavit spřažení





# Plánování v systémech reálného času

## Charakteristika RT systémů

- RT procesy **řídí** nebo **reagují** na události ve vnějším světě
- Správnost závisí nejen na **výsledku**, ale i na **čase**, ve kterém je výsledek vyprodukován
- S každou podúlohou – sdružit **deadline** – čas kdy musí být spuštěna nebo dokončena
- **Hard RT** – času **musí** být dosaženo
- **Soft RT** – dosažení deadline je **žádoucí**





# Dispatcher

## □ Dispatcher

- Modul, který předá řízení CPU procesu vybraným **short-term** plánovačem

## □ Prove:

- Přepnutí kontextu
- Přepnutí do uživatelského modu
- Skok na danou instrukci v uživatelském procesu

## □ Co nejrychlejší, vyvolán během každého přepnutí procesů

---



# Scheduler – protichůdné požadavky

- příliš časté přepínání procesu – velká režie
- málo časté – pomalá reakce systému
- čekání na diskové I/O, data ze sítě – probuzen a brzy (okamžitě) naplánován – pokles přenosové rychlosti
- multiprocesor – pokud lze, nestřídat procesory
- nastavení priority uživatelem



# Uvíznutí (deadlock)

- Příklad:
  - Naivní večeřící filozofové – vezmou levou vidličku, ale nemohou vzít pravou (už je obsazena)
  - Uvíznutí (deadlock); zablokování
-



# Uvíznutí – alokace I/O zařízení

Výhradní alokace I/O zařízení

zdroje:

Vypalovačka CD ( **V** ), scanner ( **S** )

procesy:

**A**, **B** – oba úkol naskenovat dokument a zapsat na vypalovačku

1. **A** žádá **V** a dostane, **B** žádá **S** a dostane
2. **A** žádá **S** a čeká, **B** žádá **V** a čeká -- **uvíznutí !!**



# Podmínky vzniku uvíznutí (!!!)

Coffman, 1971

## 1. vzájemné vyloučení

- Každý zdroj je buď dostupný nebo je výhradně přiřazen právě jednomu procesu

## 2. hold and wait

- Proces držící výhradně přiřazené zdroje může požadovat další zdroje
-



# Podmínky vzniku uvíznutí

## 3. nemožnost odejmutí

- Jednou přiřazené zdroje nemohou být procesu násilně odejmuty (proces je musí sám uvolnit)

## 4. cyklické čekání

- Musí být cyklický řetězec 2 nebo více procesů, kde každý z nich čeká na zdroj držený dalším členem



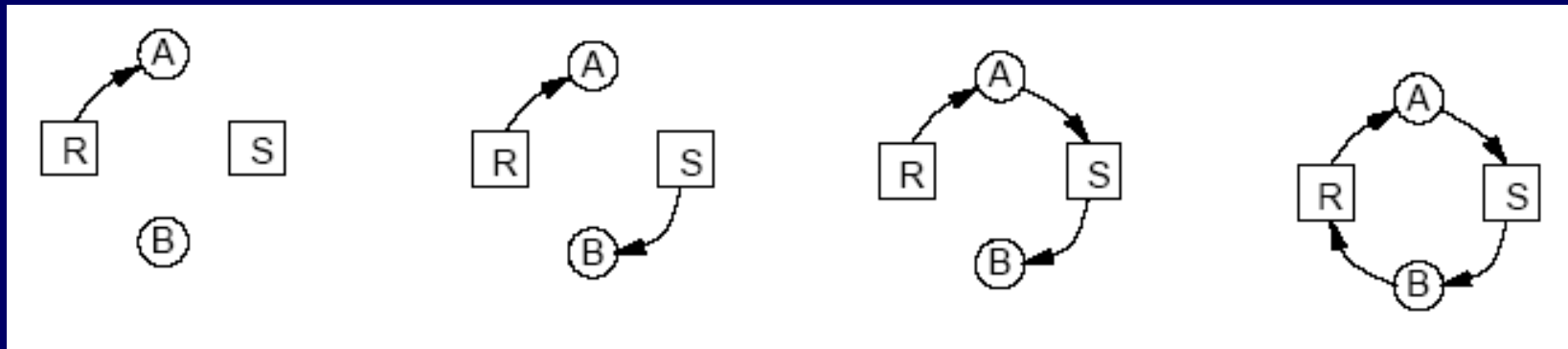
# Vznik uvíznutí - poznámky

- Pro vznik uvíznutí – musejí být **splněny všechny 4 podmínky**
  - 1. až 3. předpoklady, za nich je definována 4. podmínka
- Pokud jedna z podmínek **není splněna**, uvíznutí **nenastane**
- Viz příklad s CD vypalovačkou
  - Na CD může v jednu chvíli zapisovat pouze 1 proces
  - CD vypalovačku není možné zapisovacímu procesu odejmout

# Uváznutí

zdroje: Rekorder R a scanner S; procesy: A,B

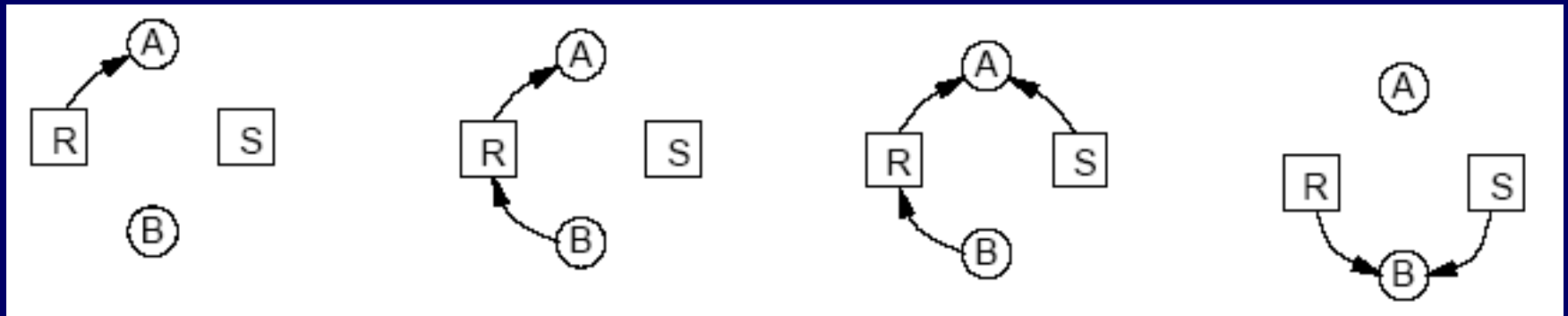
1. A žádá R dostane, B žádá S dostane
2. A žádá S a čeká, B žádá R a čeká - uváznutí





# Uvíznutí – pořadí alokace

- Pokud bychom napsali procesy A,B tak, aby oba žádaly o zdroje R a S **ve stejném pořadí** – uvíznutí **nenastane**
  1. A žádá R a dostane, B žádá R a čeká
  2. A žádá S a dostane, A uvolní R a S
  3. B čekal na R a dostane, B žádá S a dostane





# P1 – Vzájemné vyloučení

- prevence – zdroj nikdy nepřiradit **výhradně**
  - **problém lze řešit pro některé zdroje** (tiskárna)
  - **spooling**
    - pouze daemon přistupuje k tiskárně
    - nikdy nepožaduje další zdroje – není uvíznutí
    - převádí soutěžení o tiskárnu na soutěžení o diskový prostor – soutěžení o zdroj, „kterého je více“
    - pokud ale 2 procesy zaplní disk se spool souborem, žádný nemůže skončit
  - spooling není možný pro všechny zdroje (záznamy v databázi)
-



## P2- Hold and wait

- proces držící výhradně přiřazené zdroje může požadovat další zdroje
- požadovat, aby procesy **alokovaly všechny zdroje před svým spouštěním**
  - většinou nevědí, které zdroje budou chtít
  - příliš restriktivní
  - některé dávkové systémy i přes nevýhody používají, zabraňuje deadlocku
- Modifikace:  
pokud proces požaduje nové zdroje, musí **uvolnit** zdroje které drží a o všechny **požádat v jediném požadavku**



## P4 – Cyklické čekání

- Proces může mít jediný zdroj, pokud chce jiný, musí předchozí uvolnit – restriktivní, není řešení ☹
- **Všechny zdroje očíslovány, požadavky** musejí být prováděny **v číselném pořadí**
  - Alokační zdroj nemůže mít cykly
  - Problém – je těžké nalézt vhodné očíslování pro všechny zdroje
  - Není použitelné obecně, ale ve speciálních případech výhodné (jádro OS, databázový systém, ...)



# Př. Dvoufázové zamykání

- V DB systémech

- První fáze

- Zamknutí **všech** potřebných záznamů **v číselném pořadí**
- Pokud je některý zamknut jiným procesem
  - **Uvolní všechny** zámky a zkusí znovu

- Druhá fáze

- **Čtení & zápis, uvolňování zámků**

- Zamyká se vždy v číselném pořadí, uvíznutí nemůže nastat



# Shrnutí přístupu k uvíznutí (!)

1. **Ignorování problému** – většina OS ignoruje uvíznutí uživatelských procesů
2. **Detekce a zotavení** – pokud uvíznutí nastane, detekujeme a něco s tím uděláme (vrátíme čas – rollback, zrušíme proces ...)
3. **Dynamické zabránění** – zdroj přiřadíme, pouze pokud bude stav bezpečný (bankéřův algoritmus)
4. **Prevence** – strukturálně negujeme jednu z Coffman. podmínek
  - **Vzájemné vyloučení** – spooling všeho
  - **Hold and wait** – procesy požadují zdroje na začátku
  - **Nemožnost odejmutí** – odejmi (nefunguje)
  - **Cyklické čekání** – zdroje očíslovíme a žádáme v číselném pořadí



# Vyhladovění

- Procesy požadují zdroje – pravidlo pro jejich přiřazení
- Může se stát, že některý **proces zdroj nikdy neobdrží**
  - I když **nenastalo uvíznutí** !

## Př. Večeřící filozofové

- Každý zvedne levou vidličku, pokud je pravá obsazena, levou položí
- **Vyhladovění**, pokud všichni zvedají a pokládají současně



# Terminologie

- Blokovaný (blocked, waiting), někdy: čekající
    - Základní stav procesu
  
  - Uváznutí, uváznutí, deadlock, někdy: zablokování
    - Neomezené čekání na událost
  
  - Vyhladovění, starvation někdy: umoření
    - Procesy běží, ale nemohou vykonávat žádnou činnost
  - Aktivní čekání (busy wait), s předbíháním (preemptive)
-





# Jak to reálně funguje? (!!)

- ❑ proces požádá o alokaci  $n$  bajtů paměti funkcí  
ukazatel = **malloc** ( $n$ )
- ❑ malloc je knihovní fce alokátoru paměti (součást glibc)
- ❑ paměť je alokována z **haldy** (heapu) !
- ❑ alokátor se podívá, zda má volnou paměť k dispozici,  
když ne, požádá OS o přidělení dalších stránek  
paměti (systémové volání **sbrk**)
- ❑ proces uvolní paměť, když už ji nepotřebuje voláním  
free(ukazatel)

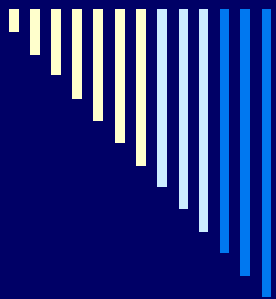


# Příklad alokace (!!)

zkuste: `man malloc`

## Příklad:

1. proces bude chtít alokovat 500B, zavolá **malloc**
2. alokátor koukne, nemá volnou paměť, požádá OS o přidělení stránky paměti (4KB) – **sbrk**
3. proces je obsloužen, dostane paměť
4. proces bude chtít dalších 200B, zavolá **malloc**
5. alokátor už má paměť v zásobě, rovnou ji přidělí procesu
6. když už proces paměť nepotřebuje, zavolá **free**



# poznámka k pointerům (!)

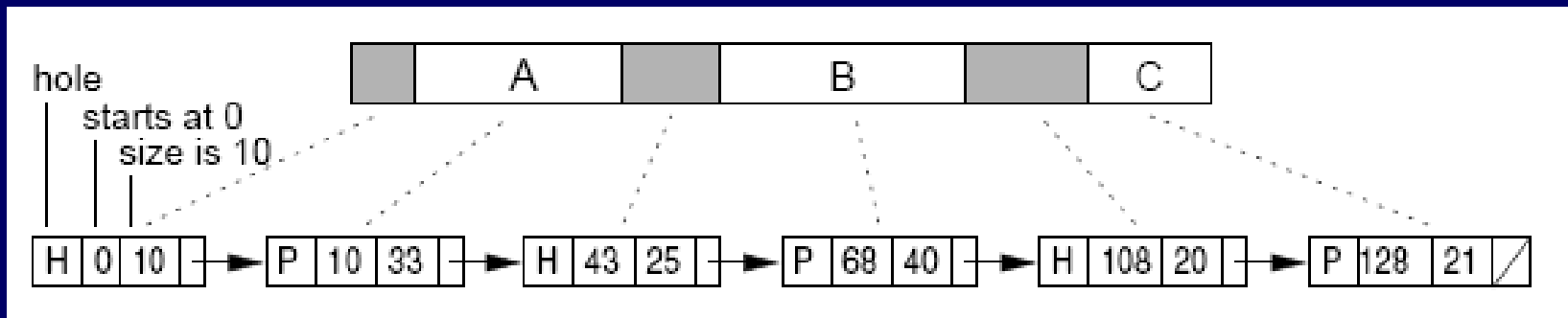
ukazatel = malloc (size)

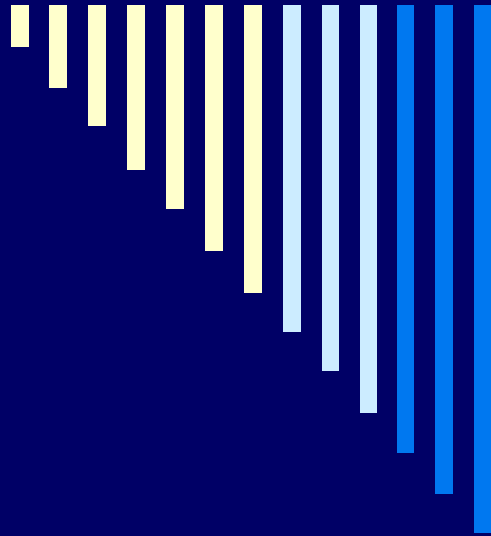
takto získaný ukazatel obsahuje **virtuální adresu**, tj. není to přímo adresa do fyzické paměti (RAM) !!!

virtuální adresa se uvnitř procesoru převede **na fyzickou adresu** (s využitím tabulky stránek atd.) !!

# Správa pomocí seznamů

- Seznam alokovaných a volných oblastí (procesů, děr)
- Položka seznamu:
  - Info o typu – proces nebo díra (P vs. H)
  - Počáteční adresa oblasti
  - Délka oblasti





# 09. Memory management

**ZOS 2014, L. Pešíčka**

---



# Poznámky

- Pro **dávkové systémy** – dosud uvedené mechanismy - přiměřené (jednoduchost, efektivita)
- **Systémy se sdílením času** – můžeme mít více procesů, než se jich **vejde** do paměti **současně**
- 2 strategie
  - **Odkládání celých procesů** (swapping)
    - Nadbytečný proces se odloží na disk
    - Např. UNIX Version 7; co platí pro velikost procesu?
  - **Virtuální paměť** – v paměti nemusí být procesy celé
    - Překrývání (overlays), virtuální paměť – bude dále



# Virtuální adresy

- fyzická paměť slouží jako cache virtuálního adresního prostoru procesů (!)
  - procesor – používá virtuální adresy
  - Pokud požadovaná část VAProstoru JE ve fyzické paměti
    - MMU převede  $VA \Rightarrow FA$ , přístup k paměti
  - požadovaná část NENÍ ve fyzické paměti
    - OS ji musí načíst z disku do RAMky
    - I/O operace – přidělení CPU jinému procesu
  - většina systémů virtuální paměti používá stránkování
-



# Pojmy – důležité !!!

- **VAP – stránky** (pages) pevné délky
    - délka mocnina 2, nejčastěji 4KB, běžně 512B - 8KB
  - **fyzická paměť – rámce** (page frames) stejné délky
  - rámec může obsahovat PRÁVĚ JEDNU stránku
  - na známém místě v paměti – **tabulka stránek**
  - tabulka stránek poskytuje mapování virtuálních stránek na rámce
-



# Stránkovaná paměť

Swap na disku



P1

P2

RAM

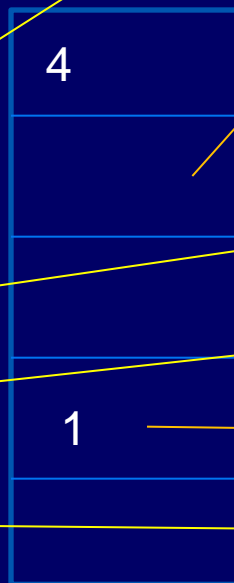
virtuální adresy

stránka např. 4KB

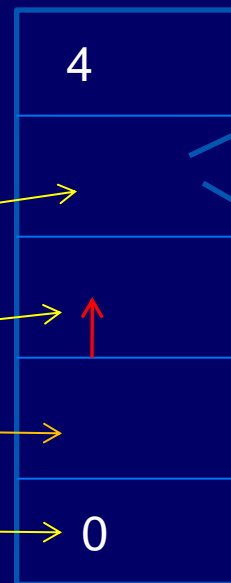
Offset od začátku stránky



Tabulka stránek  
Procesu 1



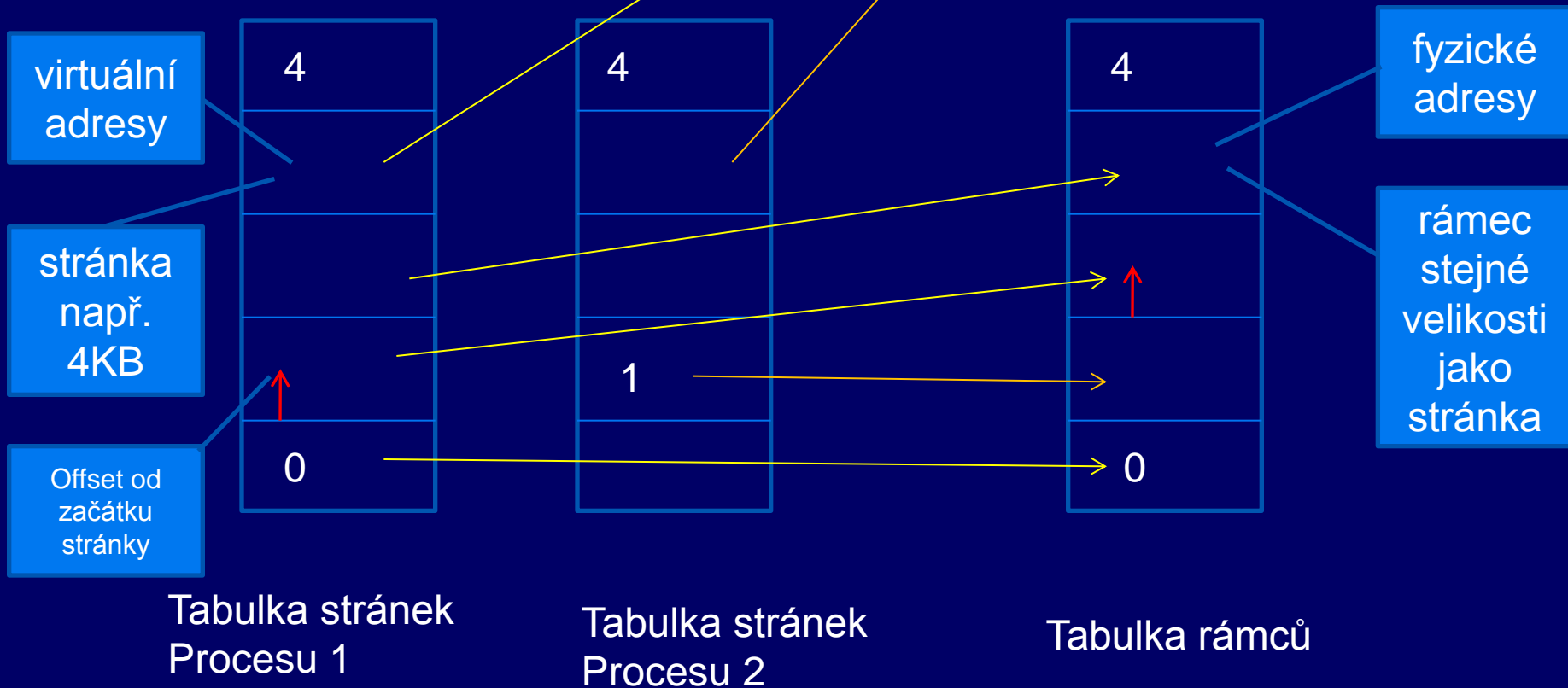
Tabulka stránek  
Procesu 2



Tabulka rámců

fyzické adresy

rámec stejné velikosti jako stránka



# Tabulka stránek - podrobněji

Číslo stránky	Číslo rámce	příznak platnosti	Příznaky ochrany	Bit modifikace (dirty)	Bit referenced	Adresa ve swapu
0	3	valid	rx	1	1	---
1	4	valid	rw	1	1	---
2	---	invalid	ro	0	0	4096

valid  
invalid

rw, rx, ro,...

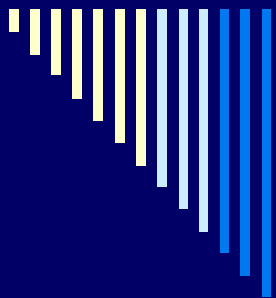
zda je třeba rámec  
uložit do swapu při  
odstranění z RAM

zda byla stránka  
přístupována (čtení či  
zápis) v poslední době



# Výpadek stránky (!!!)

- viz příklad, pro adresu 8192 str 2, offset 0
- Výpadek stránky
  - Stránka není mapována
  - Výpadek stránky způsobí **výjimku**, zachycena OS (pomocí **přerušení**)
  - OS **iniciuje zavádění** stránky a **přepne na jiný** proces
  - Po zavedení stránky OS upraví mapování (tabulku stránek)
  - Proces může pokračovat
  - Vyřešit: KAM stránku zavést a ODKUD ?



# Stránkování - poznámky

**Čisté** stránkování - bez odkládací oblasti (swapu) !!

Souvislý logický adresní prostor procesu  
**mapován do**  
nesouvislých částí paměti

OS udržuje:

- 1 tabulka rámců
- Tabulku stránek pro každý proces





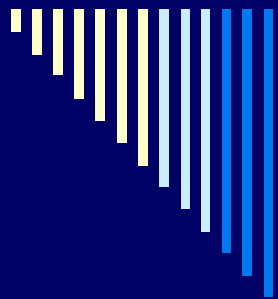
# Tabulka stránek procesu

- Mapuje číslo stránky na číslo fyzického rámce
- Další informace – např. příznaky ochrany
- Řeší problém relokační a ochrany
  - Relokace – mapování VA na FA
  - Ochrana – v tabulce stránek **pouze** stránky, ke kterým má proces **přístup**
- Přepnutí na jiný proces
  - MMU přepne na jinou tabulku stránek



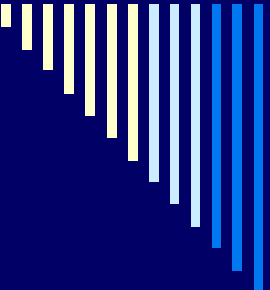
# Rychlost převodu (!)

- Každý přístup – sáhne do tabulky stránek
  - 2x více paměťových přístupů
    - musíme sáhnout do tabulky stránek a pak do paměti kam chceme
  
- TLB (Transaction Look-aside Buffer) (!!!!)
  - HW cache
  - Dosáhneme zpomalení jen 5 až 10 %
  - Přepnutí kontextu na jiný proces
    - problém (vymazání cache,..)
    - než se TLB opět zaplní – pomalý přístup



# Obsah položky v tabulce stránek (!!!)

- Číslo rámce
- Příznak platnosti (valid / invalid)
- Příznaky ochrany (rw, ro, ..)
- Bit modified (dirty)
  - zápis do stránky nastaví na 1
- Bit referenced
  - Přístup pro čtení / zápis nastaví na 1
- Další ...



# Stránkování na žádost (využívá odkládací prostor)

- Vytvoření procesu
    - Vytvoří prázdnou tabulku stránek
    - Alokace místa na disku pro odkládání stránek
    - Některé implementace – odkládací oblast inicializuje kódem programu a daty ze spustitelného souboru
  - Při běhu
    - Žádná stránka v paměti,
    - 1. přístup – **výpadek stránky (page fault)**
    - OS zavede požadovanou stránku do paměti
    - Postupně v paměti tzv. **pracovní množina** stránek
-





# Pracovní množina stránek

Má-li proces svou **pracovní množinu stránek v paměti**,  
může pracovat **bez mnoha výpadků**

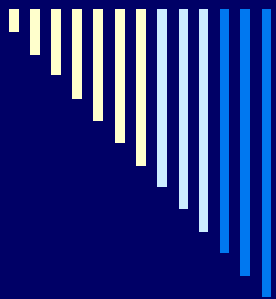
dokud se pracovní množina stránek **nezmění**, např.  
do **další fáze** výpočtu

Pracovní množina stránek daného procesu – kolik stránek  
musí mít ve fyzické paměti, aby mohl nějaký čas pracovat bez  
výpadků stránky



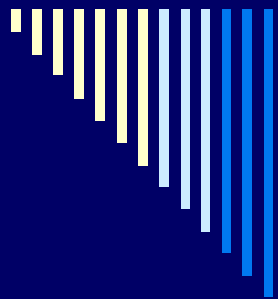
# Ošetření výpadku stránky (důležité !)

1. Výpadek – mechanismem **přerušení** (!! ) vyvolán OS
  2. OS zjistí, pro kterou stránku nastal výpadek
  3. OS určí umístění stránky na disku
    - Často tato informace přímo v tabulce stránek
  4. Najde rámec, do kterého bude stránka zavedena
    - Co když jsou všechny rámce obsazené?
  5. Načte požadovanou stránku do rámce
  6. Změní odpovídající mapovací položku v tabulce stránek
  7. Návrat..
  8. CPU provede instrukci , která způsobila výpadek
-



# Algoritmy nahrazování stránek

- Uvolnit rámec pro stránku, co s **původní stránkou**?
- Pokud byla stránka modifikována ( $\text{dirty}=1$ ), uložit na disk
- Pokud modifikovaná nebyla a má již stejnou kopii na disku (swap), pouze uvolněna



# Tabulka stránek (TS) - podrobněji

- součástí PCB (tabulka procesů) – kde leží jeho TS
- velikost záznamu v TS .. 32 bitů
- číslo rámce .. 20 bitů

dvouúrovňová tabulka stránek

- 4KB, 4MB

čtyřúrovňová tabulka stránek x86-64

- stránky 4KB, 2MB, až 1GB



# Obsah

- ❑ FIFO + Beladyho anom.
- ❑ MIN / OPT
- ❑ LRU
- ❑ NRU
- ❑ Second Chance, Clock
- ❑ Aging

-----

- ❑ Segmentování
- ❑ I/O

## Algoritmy nahrazování stránek paměti

Použijí se, pokud potřebujeme uvolnit místo v operační paměti pro další stránku:

nastal výpadek stránky, je třeba někam do RAM zavést stránku a RAM je plná..

nějakou stránku musíme z RAM odstranit, ale jakou?



# MIN / OPT

- **není realizovatelný** (křišťálová koule)
  - jak bychom zjistili dopředu která stránka bude potřeba?
- algoritmus pouze pro **srovnání** s realizovatelnými
- Použití pro běh programu v **simulátoru**
  - uchovávají se odkazy na stránky
  - spočte se počet výpadků pro MIN/OPT
  - Srovnání s jiným algoritmem (o kolik je jiný horší)



# Least Recently Used (LRU)

- **nejdéle nepoužitá** (pohled do minulosti)
- princip lokality
  - stránky používané v posledních instrukcích se budou pravděpodobně používat i v následujících
  - pokud se stránka dlouho nepoužívala, pravděpodobně nebude brzy zapotřebí
- **Vyhazovat zboží, na kterém je v prodejně nejvíce prachu = nejdéle nebylo požadováno**



# LRU – matice - příklad

reference v pořadí: 3 2 1 0

0.1.2.3						0.1.2.3						0.1.2.3						0.1.2.3					
0.	0	0	0	0	0	0.	0	0	0	0	0	0.	0	0	0	0	0	0.	0	1	1	1	1
1.	0	0	0	0	0	1.	0	0	0	0	0	1.	1	0	1	1	1	1.	0	0	1	1	1
2.	0	0	0	0	0	2.	1	1	0	1	1	2.	1	0	0	1	1	2.	0	0	0	1	1
3.	1	1	1	1	0	3.	1	1	0	0	0	3.	1	0	0	0	0	3.	0	0	0	0	0





# LRU - vlastnosti

## □ **výhody**

- z časově založených (realizovatelných) nejlepší
- Beladyho anomálie nemůže nastat

## □ **nevýhody**

- každý odkaz na stránku – aktualizace záznamu (zpomalení)
    - položka v tab. stránek
    - řádek a sloupec v matici
  - LRU se pro stránkovanou virtuální paměť příliš nepoužívá
  - LRU ale např. pro blokovou cache souborů
-



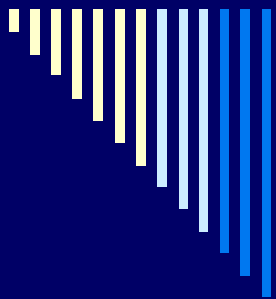
# Not-Recently-Used (NRU)

- snaha vyhazovat **nepoužívané stránky**
  - HW podpora:
    - **stavové bity Referenced (R) a Dirty (M = modified)**
    - v tabulce stránek
  - bity **nastavované HW** dle způsobu přístupu ke stránce
  - **bit R** – nastaven na 1 při **čtení** nebo **zápisu** do stránky
  - **bit M** – na 1 při **zápisu** do stránky
    - stránku je třeba při vyhození zapsat na disk
  - bit **zůstane** na 1, dokud ho SW nenastaví zpět na 0
-



# algoritmus NRU

- začátek – všechny stránky  $R=0$ ,  $M=0$
- bit **R** nastavován OS **periodicky** na 0 (přerušeni čas.)
  - odliší stránky referencované **v poslední době !!**
- **4 kategorie stránek (R,M)**
  - třída 0:  $R = 0$ ,  $M = 0$*
  - třída 1:  $R = 0$ ,  $M = 1$  -- z třídy 3 po nulování R*
  - třída 2:  $R = 1$ ,  $M = 0$*
  - třída 3:  $R = 1$ ,  $M = 1$*
- NRU vyhodí **stránku z nejnižší neprázdné třídy**
- výběr mezi stránkami ve stejné třídě je **náhodný**



# NRU

- pro NRU platí – lepší je **vyhodit modifikovanou** stránku, která **nebyla použita** 1 tik, než nemodifikovanou stránku, která se právě používá
- **výhody**
  - jednoduchost, srozumitelnost
  - efektivně implementovaný
- **nevýhody**
  - výkonnost (jsou i lepší algoritmy)



# Algoritmy Second Chance a Clock

## □ vycházejí z FIFO

- **FIFO** – obchod vyhazuje zboží zavedené před nejdelší dobou, ať už ho někdo chce nebo ne
- **Second Chance** – evidovat, jestli zboží v poslední době někdo koupil (ano – prohlásíme za čerstvé zboží)

## □ modifikace FIFO – zabránit vyhození často používané

---

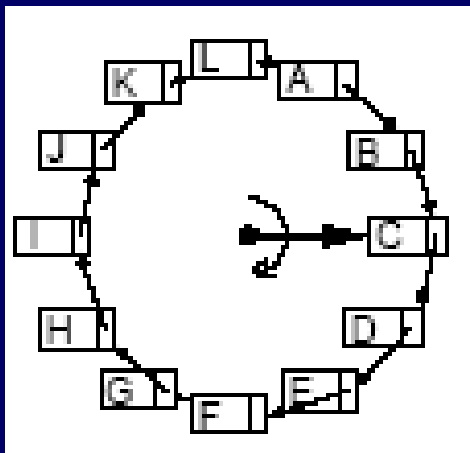


# Second Chance

- SC vyhledá **nejstarší stránku**, která **nebyla referencována v poslední době**
- Pokud všechny referencovány – **čisté FIFO**
  - Všem se postupně nastaví R na 0 a na konec seznamu
  - Dostaneme se opět na A, nyní s  $R = 0$ , vyhodíme ji
- Algoritmus končí nejvýše po (počet rámců + 1) krocích

# Algoritmus Clock

- Optimalizace datových struktur algoritmu Second Chance
  - Stránky udržovány v **kruhovém** seznamu
  - Ukazatel na nejstarší stránku – „ručička hodin“



Výpadek stránky – najít stránku k vyhození

Stránka kam ukazuje ručička

- má-li **R=0**, **stránku vyhodíme** a ručičku posuneme o jednu pozici
- má-li **R=1**, **nastavíme R na 0**, ručičku posuneme o 1 pozici, opakování,..

Od SC se liší pouze implementací

Varianty Clock používají např. BSD UNIX

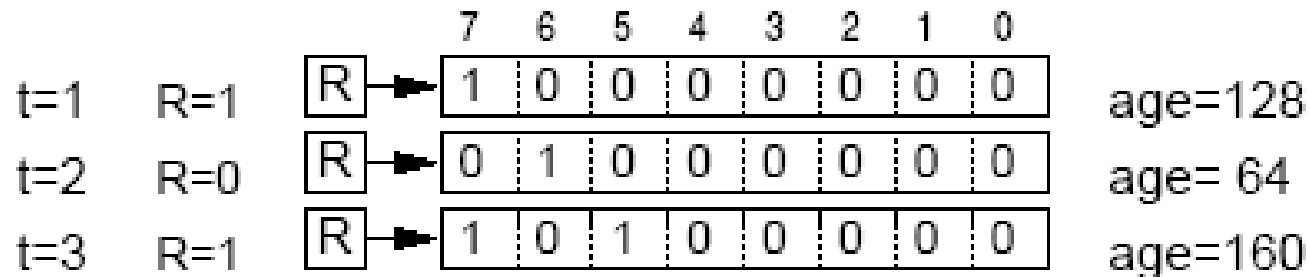


# SW aproximace LRU - Aging

- LRU vyhazuje vždy nejdéle nepoužitou stránku
- Algoritmus **Aging**
  - Každá položka tabulky stránek – pole stáří (age), N bitů (8)
  - Na počátku age = 0
  - Při každém **přerušení časovače** pro **každou** stránku:
    - Posun pole stáří o 1 bit vpravo
    - Zleva se přidá hodnota bitu R
    - Nastavení R na 0
- Při výpadku se vyhodí stránka, jejíž pole age má **nejnižší hodnotu**



# Aging



Age := age shr 1;                      posun o 1 bit vpravo

Age := age or (R shl N-1);      zleva se přidá hodnota bitu R

R := 0;                                  nastavení R na 0



# Shrnutí algoritmů

důležité je  
uvědomit si,  
kdy tyto  
algoritmy  
zafungují –  
potřebujeme  
v RAM  
uvolnit rámec

## □ Optimální algoritmus (MIN čili OPT)

- Nelze implementovat, vhodný pro srovnání

## □ FIFO

- Vyhazuje nejstarší stránku
- Jednoduchý, ale je schopen vyhodit důležité stránky
- Trpí Beladyho anomálií

## □ LRU (Least Recently Used)

- Výborný
- Implementace vyžaduje spec. hardware, proto používán zřídka



# Shrnutí algoritmů II.

## □ NRU (Not Recently Used)

- Rozděluje stránky do 4 kategorií dle bitů R a M
- Efektivita není příliš velká, přesto používán

## □ Second Chance a Clock

- Vycházejí z FIFO, před vyhozením zkontrolují, zda se stránka používala
- Mnohem lepší než FIFO
- Používané algoritmy (některé varianty UNIXu)

## □ Aging

- Dobře aproximuje LRU – efektivní
  - Často prakticky používaný algoritmus
-



# Lokální alokace

- Kolik rámců dát každému procesu?
  - **Nejjednodušší** – všem procesům dát stejně
    - Ale potřeby procesů jsou různé
  - **Proporcionální** – každému proporcionální díl podle velikosti procesu
  - **Nejlepší** – podle frekvence výpadků stránek (Page Fault Frequency, PFF)
    - Pro většinu rozumných algoritmů se PFF snižuje s množstvím přidělených rámců
-



# Zloděj stránek (page daemon)

- v systému se běžně udržuje určitý počet volných rámců
- když klesne pod určitou mez, pustí **page daemon - kswapd** (zloděj stránek), ten uvolní určité množství stránek (rámců)
- když se čerstvě uvolněné stránky hned nepřidělí, lze je v případě potřeby snadno vrátit příslušnému procesu



# Mechanismus VP - výhody

## □ Rozsah virtuální paměti

- (32bit: 2GB pro proces + 2GB pro systém, nebo 3+1)
- Adresový prostor úlohy není omezen velikostí fyzické paměti
- Multiprogramování – není omezeno rozsahem fyz. paměti

## □ Efektivnější využití fyzické paměti

- Není vnější fragmentace paměti
  - Nepoužívané části adresního prostoru úlohy nemusejí být ve fyzické paměti
-



# Mechanismus VP - nevýhody

- Režie při převodu virt. adres na fyzické adresy
- Režie procesoru
  - údržba tabulek stránek a tabulky rámců
  - výběr stránky pro vyhození, plánování I/O
- Režie I/O při čtení/zápisu stránky
- Paměťový prostor pro tabulky stránek
  - Tabulky stránek v RAM, často používaná část v TLB
- Vnitřní fragmentace
  - Přidělená stránka nemusí být plně využita

# Rozdělení paměti pro proces (!!!)

pokus.c:

```
int x =5; int y = 7; // inic. data
```

```
void fce1() {  
    int pom1, pom2; // na zásobníku  
    ... }
```

```
int main (void) {  
    ...  
    malloc(1000); // halda  
    fce1();  
    return 0;  
}
```



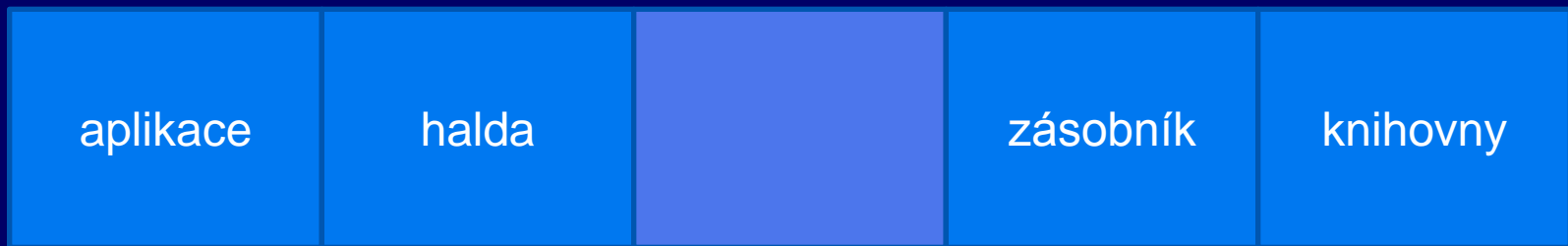


# Rozdělení paměti pro proces

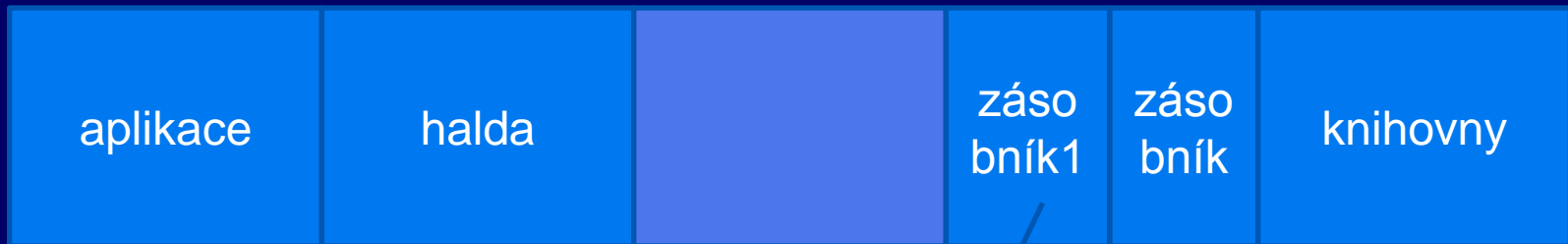
Roste halda



Roste zásobník



Máme-li více vláken => více zásobníků, limit velikosti zásobníku



zásobník dalšího vlákna





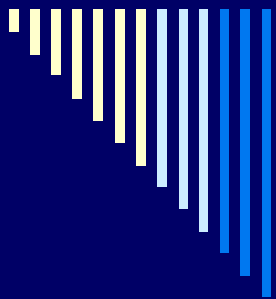
# Segmentace

- Dosud diskutovaná VP – **jednorozměrná**
  - Proces: adresy **< 0, maximální virtuální adresa>**
- Často výhodnější – **více samostatných virtuálních adresových prostorů**
- Př. – máme několik tabulek a chceme, aby jejich velikost mohla růst
- Paměť nejlépe více nezávislých adresových prostorů - **segmenty**



# Čistá segmentace

- Každý odkaz do paměti – dvojice (**selektor, offset**)
    - **Selektor** – číslo segmentu, určuje segment
    - **Offset** – relativní adresa v rámci segmentu
  - Technické prostředky musí umět přemapovat dvojici (selektor, offset) na **lineární adresu** (fyzická když není dále stránkování)
  - **Tabulka segmentů** – každá položka má
    - Počáteční adresa segmentu (**báze**)
    - Rozsah segmentu (**limit**)
    - Příznaky ochrany segmentu (čtení, zápis, provádění – rwx)
-



# Segmentace se stránkováním

- velké segmenty – nepraktické celé udržovat v paměti
- Myšlenka stránkování segmentů
  - V paměti pouze potřebné stránky
- Implementace – např. každý segment vlastní tabulka stránek



# Adresy (!!!)

virtuální adresa -> lineární adresa -> fyzická adresa

virtuální – používá proces

lineární – po segmentaci  
pokud není dále stránkování, tak už  
představuje i fyzickou adresu

fyzická – adresa do fyzické paměti RAM  
(CPU jí vystaví na sběrnici)

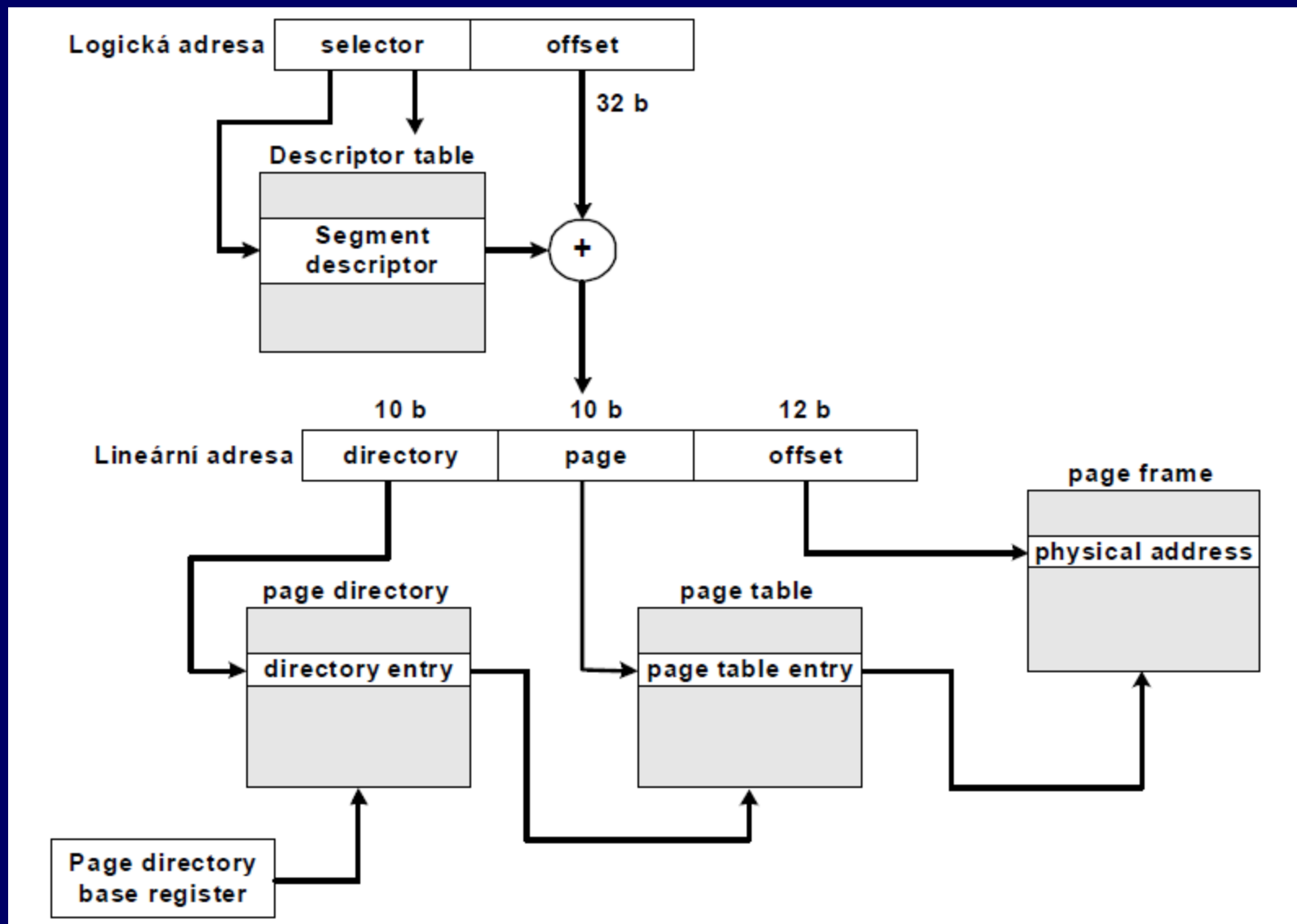
---



# Selektor segmentu

- 13 bitů – index, tj.  $\max 2^{13} = 8192$  položek
  - selektor 0 – indikace nedostupnosti segmentu
  
- při zavedení selektoru do segmentového registru CPU také zavede odpovídající popisovač z LDT nebo GDT do vnitřních registrů CPU
  - bit 2 selektoru – pozná, zda LDT nebo GDT
  - popisovač segmentu na adrese (selektor and 0fff8h) + zač. tabulky

# Komplexní schéma převodu VA na FA (pamatovat !!!!)





# Procesory & přerušení

## □ reálný mód

- První kilobyte (0..1023) RAM
  - interrupt vector table (256x4B)

## □ chráněný mód

- IDT (Interrupt Descriptor Table)
  - pole 8bytových deskriptorů (indexovaných přerušovacím vektorem)
  - naplněná IDT tabulka 2KB (256x8B)
-





# Pamatuj !

- ❑ Běžný procesor v PC může běžet v reálném nebo chráněném módu
- ❑ Po zapnutí napájení byl puštěn **reálný mód**, ten využíval např. MS-DOS – není zde však žádný mechanismus ochrany
- ❑ Dnešní systémy přepínají procesor ihned do **chráněného režimu** (ochrana segmentů uplatněním limitu, ochrana privilegovanosti kontrolou z jakého ringu je možné přistupovat)



# Chráněný režim - segmenty

- 1 GDT – může mít až 8192 segmentů
- můžeme mít i více LDT (každá z nich může mít opět 8192 segmentů) a použít je pro ochranu procesů
- některé systémy využívají jen GDT, a místo LDT zajišťují ochranu pomocí stránkování



# Chráněný režim – adresy !!!!

**VA(selektor,offset) =segmentace==> LA =stránkování==> FA**

**VA** je virtuální adresa, **LA** lineární adresa, **FA** fyzická adresa  
**selektor** určí odkaz do tabulky segmentů => **deskriptor** (v GDT nebo LDT)

selektor obsahuje mj. bázi a limit ; **LA = báze + offset**

segmentaci nejde vypnout, stránkování ano

zda je zapnuté stránkování - bit v řídicím registru procesoru (**CR0**)

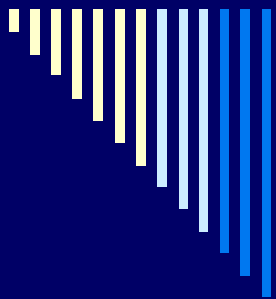
je-li vypnuté stránkování, lineární adresa představuje fyzickou adresu

chce-li systém používat jen stránkování,

roztáhne segmenty přes celý adresní prostor (překrývají se)

Linux: využívá stránkování, Windows: obojí (ale viz předchozí komentáře)

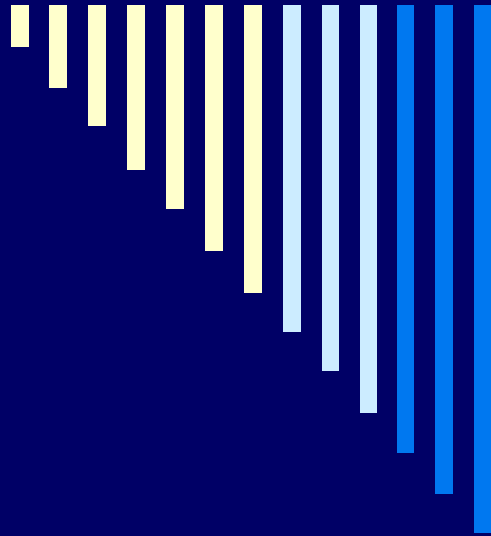
---



# Poznámky - prepaging

Prepaging:

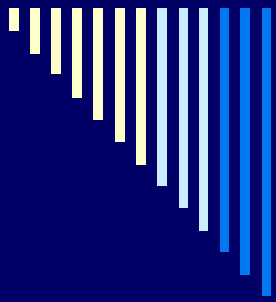
do paměti se zavádí chybějící stránka a stránky okolní



# 11., 12. Správa I/O Správa souborů

**ZOS 2014, L. Pešička**

---



# Vývoj rozhraní mezi CPU a zařízeními

1. CPU řídí přímo periferii
2. CPU – řadič – periférie  
aktivní čekání CPU na dokončení operace
3. řadič umí vyvolat přerušení
4. řadič umí DMA
5. I/O modul
6. I/O modul s vlastní pamětí



# 1. CPU řídí přímo periférii

- ❑ CPU přímo vydává potřebné signály
  - ❑ CPU dekoduje signály poskytovaném zařízením
  - ❑ Nejjednodušší HW
  - ❑ Nejméně efektivní využití CPU
- 
- ❑ Jen v jednoduchých mikroprocesorem řízených zařízeních (dálkové ovládání televize)



## 2. CPU – řadič - periférie

### Řadič (device controller)

- Převádí příkazy CPU na elektrické impulzy pro zařízení
  - Poskytuje CPU info o stavu zařízení
  - Komunikace s CPU pomocí **registrů** řadiče na známých I/O adresách
  - HW buffer pro alespoň 1 záznam (blok, znak, řádka)
  - Rozhraní řadič-periférie může být standardizováno (SCSI, IDE, ...)
-





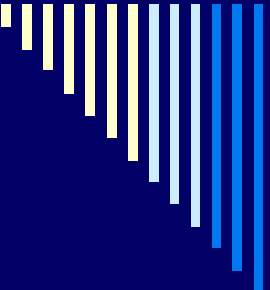
## 2. řadič – příklad operace zápisu

- CPU zapíše data do bufferu,  
Informuje řadič o požadované operaci
- Po dokončení výstupu zařízení nastaví příznak, který  
může CPU otestovat
- if přenos == OK, může vložit další data
- CPU musí dělat všechno (programové I/O)
- Významnou část času stráví CPU čekáním na  
dokončení I/O operace



### 3. Řadič umí vyvolat přerušení

- CPU nemusí testovat příznak dokončení
- Při dokončení I/O vyvolá řadič **přerušení**
- CPU začne obsluhovat přerušení
  - Provádí instrukce na předdefinovaném místě
  - Obslužná procedura přerušení
  - Určí co dál
- Postačuje pro pomalá zařízení, např. sériové I/O



## 4. Řadič může přistupovat k paměti pomocí DMA

- ❑ DMA přenosy mezi pamětí a buffery
  - ❑ CPU vysílá příkazy,  
při přerušení analyzuje status zařízení
  - ❑ CPU inicializuje přenos, ale sám ho nevykonává
  - ❑ Bus mastering – zařízení převezme kontrolu nad sběrnici a přenos provede samo (PCI sběrnice)
  - ❑ Vhodné pro rychlá zařízení – řadič disků, síťová karta, zvuková karta, grafická karta atd.
-



## 5. I/O modul umí interpretovat speciální I/O programy

- ❑ I/O procesor
- ❑ Interpretuje programy **v hlavní paměti**
- ❑ CPU spustí I/O procesor  
I/O procesor provádí své instrukce samostatně



## 6. I/O modul s vlastní pamětí

- I/O modul provádí programy
- Má **vlastní paměť**(!)
  - Je vlastně **samostatným počítačem**
- Složité a časově náročné operace  
grafika, šifrování, ...



# Komunikace CPU s řadičem

- **Odlišné adresní prostory**
    - CPU zapisuje do registrů řadiče pomocí speciálních I/O instrukcí
    - Vstup: **IN** R, port
    - Výstup: **OUT** R, port
  - **1 adresní prostor**
  - **Hybridní schéma**
-



# RAID

- pevný disk
    - elektronická část + mechanická
    - náchylost k poruchám
    - cena dat >> cena hw
  - odstávka při výměně zařízení
    - náhrada hw, přenos dat ze zálohy - prostoje
    - SLA 24/7
  - větší disková kapacita než 1 disk
  - RAID
    - Redundant Array of Independent (Inexpensive) disks
-

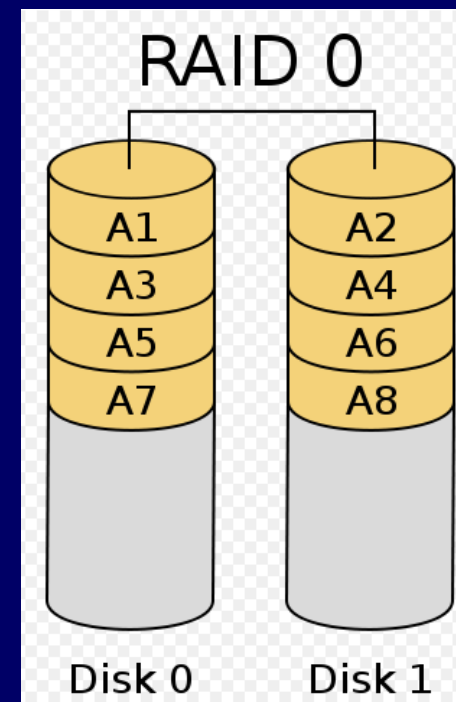
# RAID 0

## □ Zřetězení

- Data postupně ukládána na několik disků
- Zaplní se první disk, pak druhý, atd.
- Snadné zvětšení kapacity, při poruše disku ztratíme jen část dat

## □ Prokládání

- Data ukládána na disky cyklicky po blocích (stripy)
- Při poruše jednoho z disků – přijdeme o data
- Větší rychlost čtení / zápisu
  - Jeden blok z jednoho disku, druhý blok z druhého disku



Na obrázku je režim prokládání, zdroj: wikipedia (i u dalších obrázků)





# RAID 1

- mirroring .. zrcadlení
- na 2 disky stejných kapacit totožné informace
- výpadek 1 disku – nevadí
- jednoduchá implementace – často čistě sw
- nevýhoda – využijeme jen polovinu kapacity
- zápis – pomalejší (stejná data na 2 disky)  
ovlivněn diskem, na němž bude trvat déle
- čtení – rychlejší  
(řadič - lze střídat požadavky mezi disky)



# RAID 5

- redundantní pole s distribuovanou paritou
- minimálně 3 disky
- režie: 1 disk z pole  $n$  disků
  - 5 disků 100GB : 400GB pro data
- výpadek 1 disku nevadí
- čtení – výkon ok
- zápis – pomalejší
  - 1 zápis – čtení starých dat, čtení staré parity, výpočet nové parity, zápis nových dat, zápis nové parity



# RAID 6

- ❑ RAID 5 + navíc další paritní disk
- ❑ odolné proti výpadku dvou disků
- ❑ Rychlost čtení srovnatelná s RAID 5
- ❑ Zápis pomalejší

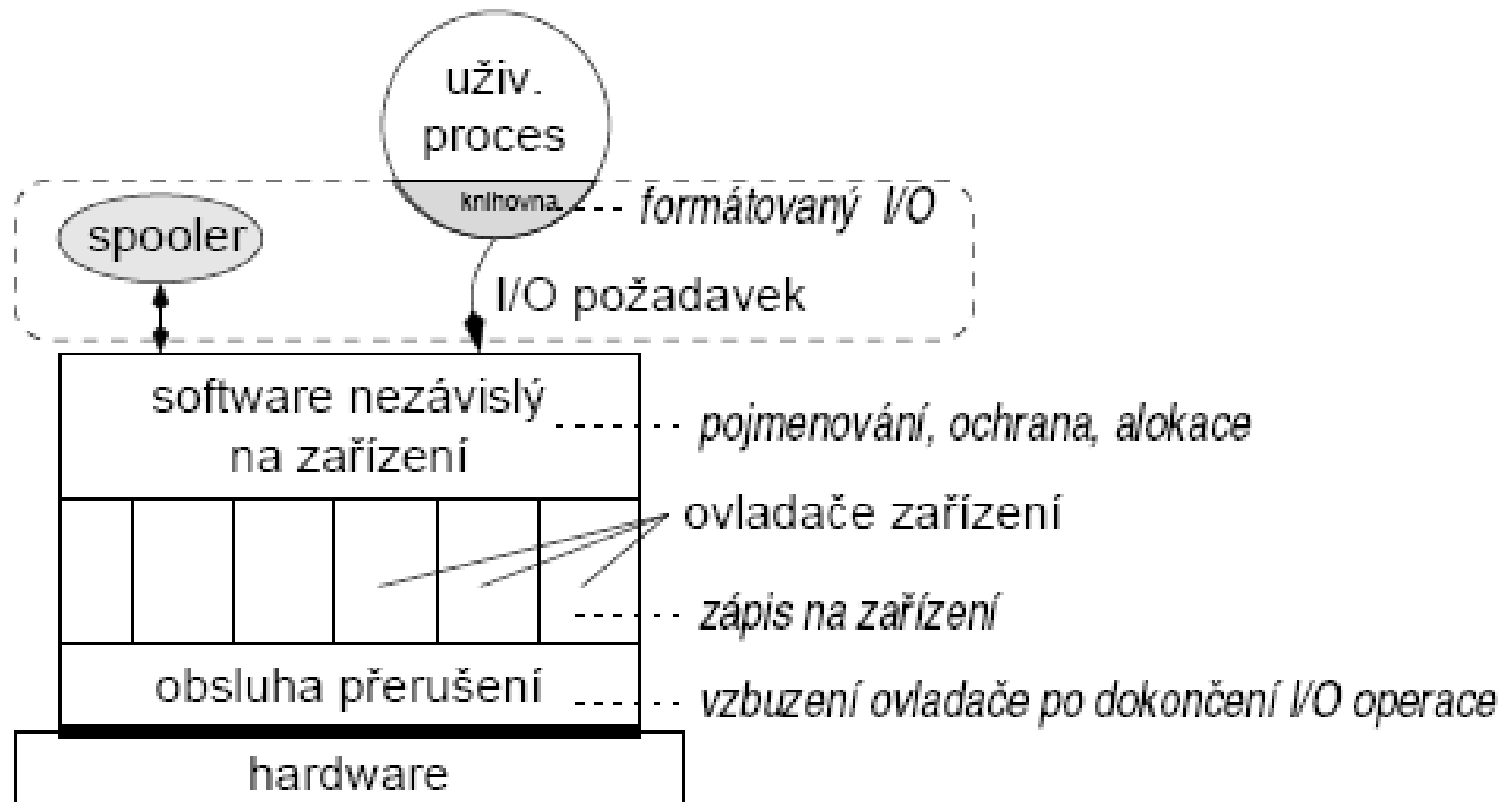


# Principy I/O software (!!!!)

typicky strukturován do 4 úrovní:

1. obsluha přerušení (nejnižší úroveň v OS)
2. ovladač zařízení
3. SW vrstva OS nezávislá na zařízení
4. uživatelský I/O SW

Toto je potřeba znát !!





# 1. Obsluha přerušení

- ❑ řadič vyvolá přerušení ve chvíli **dokončení** I/O požadavku
  - ❑ snaha, aby se přerušením nemusely zabývat vyšší vrstvy
  - ❑ ovladač zadá I/O požadavek, **usne** - P(sem)
  - ❑ po příchodu přerušení ho obsluha přerušení **vzbudí** - V(sem)
  - ❑ časově kritická obsluha přerušení – co nejkratší
-



## 2. Ovladače zařízení

- obsahují veškerý kód závislý na konkrétním I/O zařízení (např. zvukovka od daného výrobce)
  - ovladač zná jediný hw podrobnosti
    - způsob komunikace s řadičem zařízení
    - zná detaily – např. ví o sektorech a stopách na disku, pohybech diskového raménka, start & stop motoru
  - může ovládat všechna zařízení daného druhu nebo třídu příbuzných zařízení
    - např. ovladač SCSI disků – všechny SCSI disky
-



# Funkce ovladače zařízení

- ovladači předán příkaz od vyšší vrstvy
    - např. zapiš data do bloku n
  - nový požadavek **zařazen do fronty**
    - může ještě obsluhovat předchozí
  - ovladač zadá **příkazy řadiči** (požadavek přijde na řadu)
    - např. nastavení hlavy, přečtení sektoru
  - **zablokuje se do vykonání** požadavku
    - neblokuje při rychlých operacích – např. zápis do registru
  - **vzbuzení** obsluhou přerušení (dokončení operace) – zkontroluje, zda nenastala chyba
-





### 3. SW vrstva OS nezávislá na zařízení

- I/O funkce společné pro všechna zařízení daného druhu
  - např. společné fce pro všechna bloková zařízení
- definuje rozhraní s ovladači
- poskytuje jednotné rozhraní uživatelskému SW
  
- viz další slide...



# Poskytované funkce (!)

- pojmenování zařízení
    - LPT1 x /dev/lp0
  - ochrana zařízení ( přístupová práva)
  - alokace a uvolnění vyhrazených zařízení
    - v 1 chvíli použitelná pouze jedním procesem
    - např. tiskárna, plotter, magnetická páska
  - vyrovnávací paměti
    - bloková zařízení – bloky pevné délky
    - pomalá zařízení – čtení / zápis s využitím bufferu
-



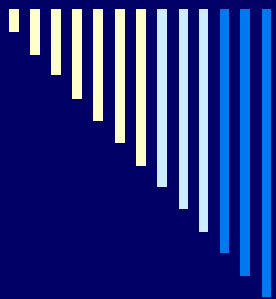
## 4. I/O sw v uživatelském režimu

- programátor používá v programech **I/O funkce** nebo **příkazy** jazyka
    - např. printf v C, writeln v Pascalu
    - knihovny sestavené s programem
    - formátování - printf("%.2d:%.2d\n", hodin, minut)
    - často vlastní vyrovnávací paměť na jeden blok
  - **spooling**
    - implementován pomocí procesů běžících v uživ. režimu
    - způsob obsluhy vyhrazených I/O zařízení (multiprogram.)
    - např. proces by alokoval zařízení a pak hodinu nic nedělal
-



# Souborové systémy

- potřeba aplikací **trvale** uchovávat data
- **hlavní požadavky**
  - možnost uložit velké množství dat
  - informace zachována i po ukončení procesu
  - data přístupná více procesům
- **společné problémy** při přístupu k zařízení
  - alokace prostoru na disku
  - pojmenování dat
  - ochrana dat před neoprávněným přístupem
  - zotavení po havárii (výpadek napájení)



# Soubor

- OS pro přístup k médiím poskytuje abstrakci od fyzických vlastností média – soubor
- soubor = pojmenovaná množina souvisejících informací
- souborový systém (file system, fs)
  - konvence pro ukládání a přístup k souborům
    - datové struktury a algoritmy
  - část OS, poskytuje mechanismus pro ukládání a přístup k datům, implementuje danou konvenci



# Základní znalosti

`fdisk /dev/sda`

- ☐ Zobrazení rozdělení disku na oddíly (partitions)
- ☐ Možnost toto rozdělení změnit

`/sbin/mkfs.ext4 /dev/sda1`

- ☐ Zformátování oddílu na vybraný filesystem (zde ext4)



# Počet oddílů

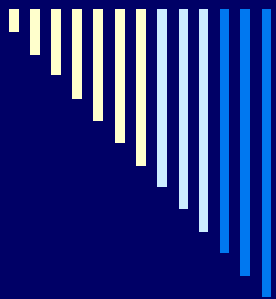
## 2 způsoby dělení

- **Master Partition Table (MPT)**

- ▣ Master Boot Record (MBR) – na počátku disku
- ▣ Umožňuje **4 oddíly primární**
- ▣ Chceme-li více, **3 primární a 1 extended**, který lze dělit na další oddíly

- **GUID Partition Table (GPT)**

- ▣ Nelimituje na 4 oddíly, např. Microsoft 124 oddílů
  - ▣ Používá např. Mac OS
-



# Vnitřní struktura (obyčejného) souboru

## □ 3 časté způsoby

- nestrukturovaná posloupnost bytů
- posloupnost záznamů
- strom záznamů

## □ nestrukturovaná posloupnost bytů (nejčastěji)

- OS obsah souboru nezajímá, interpretace je na aplikacích
- maximální flexibilita
  - programy mohou strukturovat, jak chtějí





# Paměťově mapované soubory (!!)

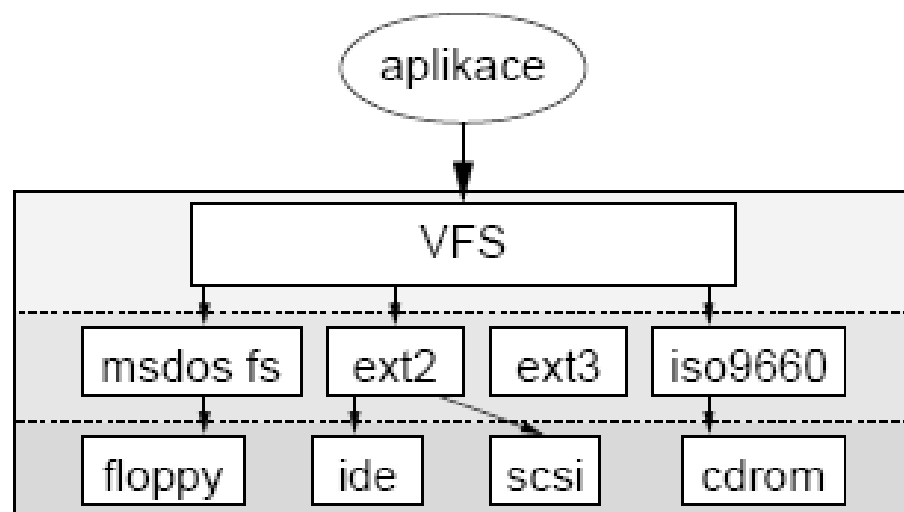
- někdy se může zdát open/read/write/close nepohodlné
- možnost mapování souboru do adresního prostoru procesu
- služby systému `mmap()`, `munmap()`
- mapovat je možné i jen část souboru
- k souboru pak přistupujeme např. přes pointery v C



# Adresářová struktura

- Jeden oddíl (partition) disku obsahuje jeden fs
- fs – 2 součásti:
  - množina souborů, obsahujících **data**
  - **adresářová struktura** – udržuje informace o všech souborech v daném fs
- adresář **překládá** jméno souboru na informace o souboru (umístění, velikost, typ ...)

# Implementace souborových systémů (!!!)



logický souborový systém

moduly organizace souborů

ovladače zařízení



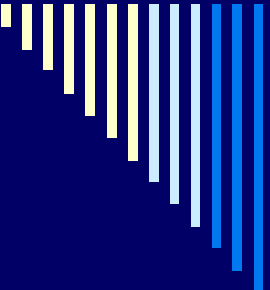
# Implementace fs - vrstvy

1. **Logický (virtuální) souborový systém**
    - ▣ Volán aplikacemi
  2. **Modul organizace souborů**
    - ▣ Konkrétní souborový systém (např. ext3)
  3. **Ovladače zařízení**
    - ▣ Pracuje s daným zařízením
    - ▣ Přečte/zapíše logický blok
-



# Ad 1 – virtuální fs

- Volán aplikacemi
  - Rozhraní s moduly organizace souborů
  - Obsahuje kód **společný** pro všechny typy fs
  - **Převádí** jméno souboru na informaci o souboru
  - **Udržuje informaci** o otevřeném souboru
    - Pro čtení / zápis (režim)
    - Pozice v souboru
  - **Ochrana a bezpečnost** (ověřování přístupových práv)
-



## Ad 2 – modul organizace souborů

- Implementuje **konkrétní** souborový systém
    - ext3, xfs, ntfs, fat, ..
  - **Čte/zapisuje datové bloky** souboru
    - Číslovány 0 až N-1
    - Převod čísla bloku na diskovou adresu
    - Volání ovladače pro čtení – zápis bloku
  - **Správa volného** prostoru + **alokace** volných bloků
  - Údržba datových struktur filesystemu
-

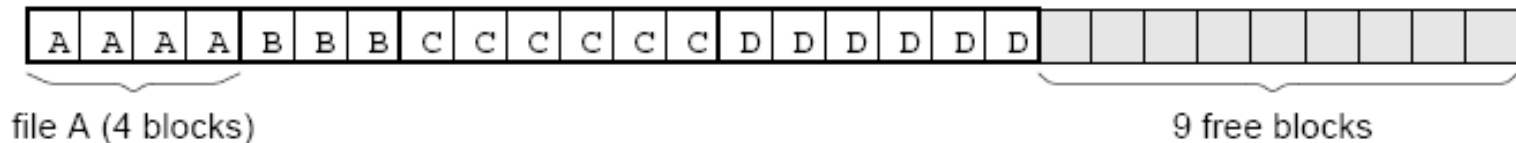


## Ad 3 – ovladače zařízení

- Nejnižší úroveň
- Floppy, ide, scsi, cdrom
- Interpretují požadavky:  
přečti logický blok 6456 ze zařízení 3

# Kontinuální alokace

- Soubor jako **kontinuální posloupnost** diskových bloků
- Příklad: bloky velikosti 1KB, soubor A (4KB) by zabíral 4 po sobě následující bloky
- Implementace
  - Potřebujeme znát číslo prvního bloku
  - Znat celkový počet bloků souboru (např. v adresáři)
- **Velmi rychlé čtení**
  - Hlavičku disku na začátek souboru, čtené bloky jsou za sebou







# Lze dnes využít kontinuální alokaci?

- Dnes se používá pouze na read-only a write-once médiích
- Např. v ISO 9660 pro CD ROM



# FAT (!!)

- Přesunutí odkazů do samostatné tabulky FAT
- **FAT (File Allocation Table)**
  - Každému diskovému bloku **odpovídá** jedna položka ve FAT tabulce
  - Položka FAT obsahuje číslo **dalšího bloku** souboru (je zároveň odkazem **na další položku** FAT!)
  - Řetězec odkazů je ukončen speciální značkou, která není platným číslem bloku

## Tabulka FAT

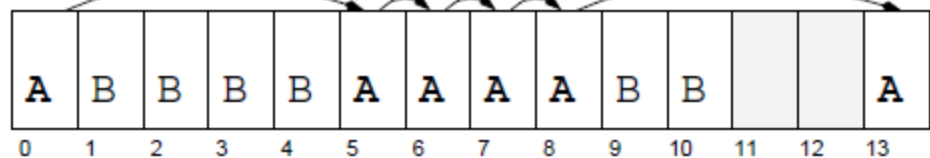
Položka č.: 0	5	
1	2	
2	3	
3	4	
4	9	
5	6	
6	7	
7	8	
8	13	
9	10	
10	-1	
11	volný	
12	volný	
13	-1	

--- soubor A začíná zde  
 --- soubor B začíná zde  
 --- značka konce souboru (EOF marker)

Fakt tu je!  
Jdem dál na  
6

5 znamená, že na indexu 5  
je další odkaz na blok  
souboru A

Na indexu 5 je i datový  
blok souboru A (kus filmu)

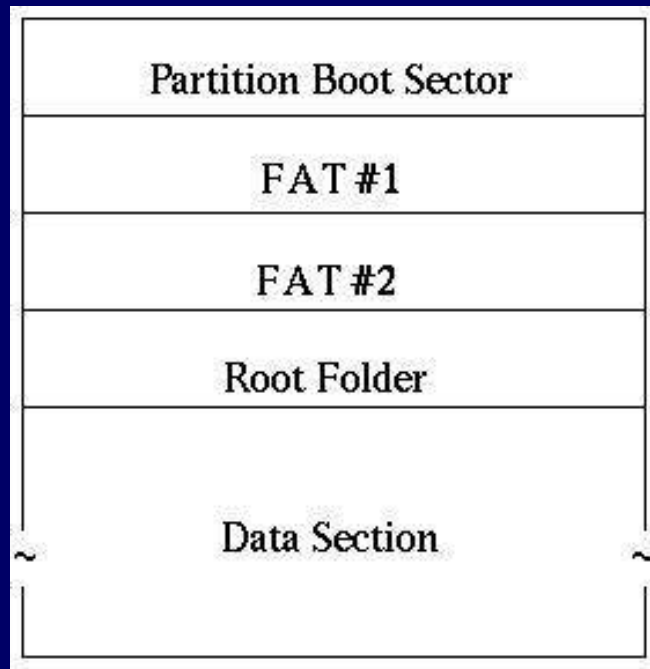


datové bloky na disku

Co je na FAT důležité?

Pokud bych chtěl např. k souboru B přidat další datový blok, nemusím s ničím hýbat, pouze do FAT(10) vložím číslo 11, a do FAT(11) dám -1 a soubor B je prodloužený

# Diskový oddíl s FAT



Diskový oddíl, který je naformátován na FAT (např. FAT32)

1. boot sektor (pokud se z daného oddílu bootuje)
2. 2 kopie FAT tabulky
3. hlavní adresář daného oddílu
4. data

Zdroj obrázku:

<http://yliqepyh.keep.pl/fat-file-system-specifications.php>



# Příklady filesystemů (!!!)

## □ **FAT**

- Starší verze Windows, paměťové karty
- Nepoužívá ACL – u souborů není žádná info o přístupových právech
- snadná přenositelnost dat mezi různými OS

## □ **NTFS**

- Používá se ve Windows XP/Vista/7
- Používají ACL: k souboru je přiřazen seznam uživatelů, skupin a jaká mají oprávnění k souboru (!!!!)

## □ **Ext2**

- Použití v Linuxu, nemá žurnálování
- Nepoužívá ACL – jen standardní nastavení (vlastní, skupina, others), což ale není ACL (to je komplexnější)

## □ **Ext3**

- Použití v Linuxu, má žurnál (rychlejší obnova konzistence po výpadku)
-



# NTFS – způsob uložení dat (!!!)

- kódování délkou běhu
- od pozice 0 máme např. uloženo:  
A1, A2, A3, B1, B2, A4, A5, C1, ...
- soubor A bude popsán fragmenty
- **fragment**
  - index
  - počet bloků daného souboru
- v našem příkladě:
  - 0, 3 (od indexu 0 patří tři bloky souboru A)
  - 5, 2 (od indexu 5 patří dva bloky souboru A)

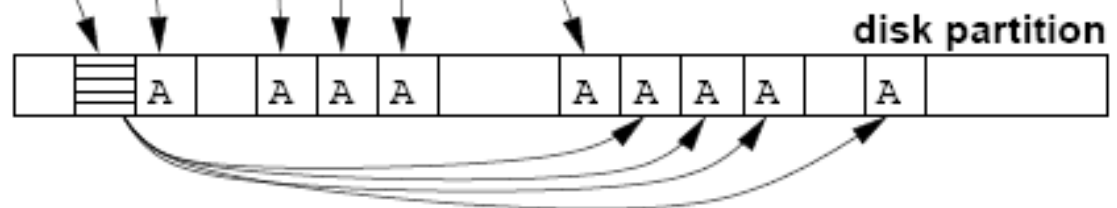


# I-uzly (!!)

- S každým souborem sdružena datová struktura i-uzel (i-node, zkratka z index-node)
  - i-uzel obsahuje
    - Atributy souboru
    - Diskové adresy prvních N bloků
    - 1 či více odkazů na diskové bloky obsahující další diskové adresy (případně obsahující odkazy na bloky obsahující adresy)
  - Používá tradiční fs v Unixu UFS (Unix File System) a z něj vycházející v Linuxu, dnes např. ext2, ext3, ext4
-

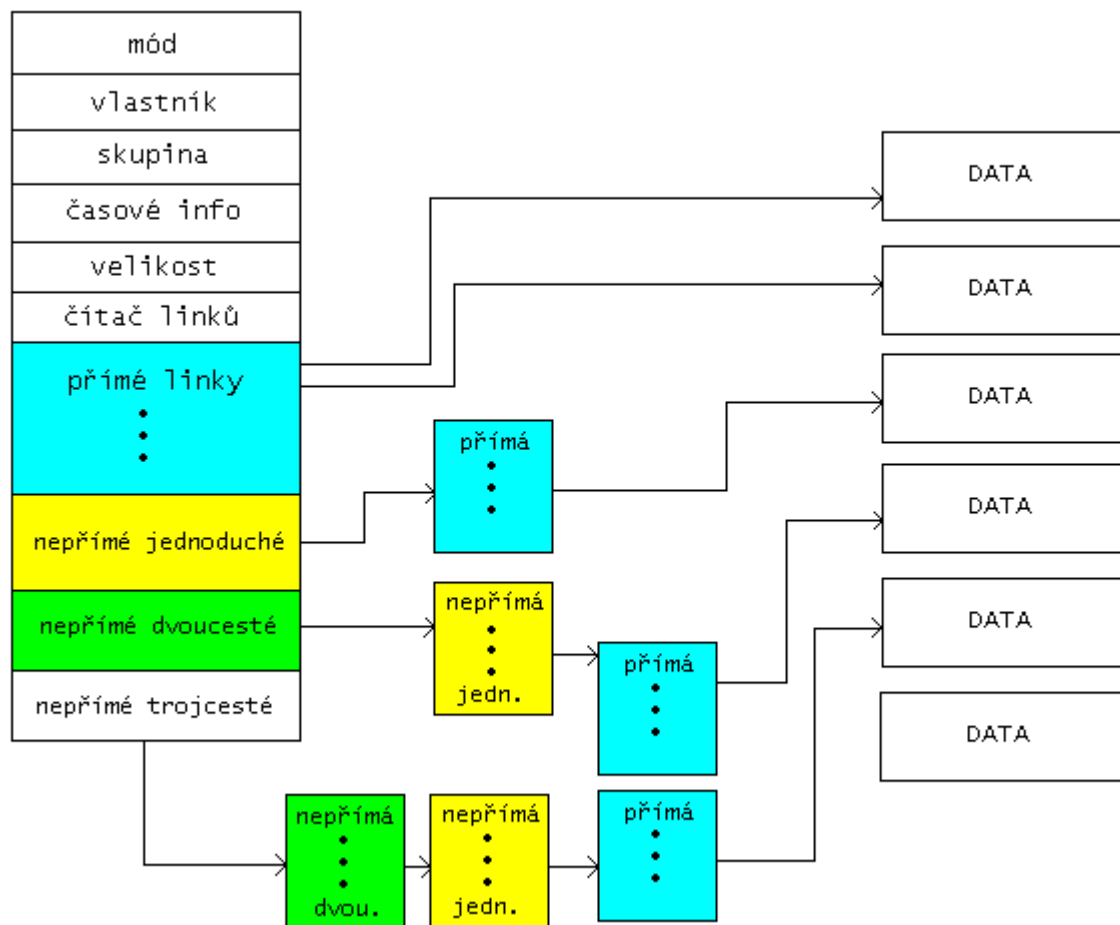
### i-node

attributes
pointer to block 0
pointer to block 1
pointer to block 2
pointer to block 3
...
pointer to block 10
pointer to block of pointers





# i-uzly dle normy POSIX



zdroj: <http://cs.wikipedia.org/wiki/Inode>



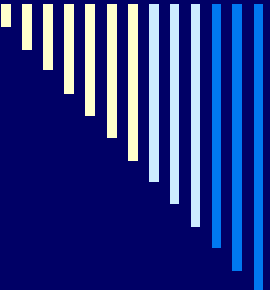
# i-uzly dle normy Posix

- ❑ **MODE** – typ souboru, **přístupová práva** (u,g,o)
  - ❑ **REFERENCE COUNT** – počet odkazů na tento objekt  
(vytvoření hardlinku zvyšuje počet)
  - ❑ **OWNER** – ID vlastníka
  - ❑ **GROUP** – ID skupiny
  - ❑ **SIZE** – velikost objektu
  - ❑ **TIME STAMPS**
    - atime – čas posledního přístupu (čtení souboru, výpis adresáře)
    - mtime – čas poslední změny
    - ctime – čas poslední změny i-uzlu (metadat)
-

# i-uzly dle normy POSIX

- ❑ **DIRECT BLOCKS** – 12 přímých odkazů na datové bloky (data v souboru)
- ❑ **SINGLE INDIRECT** – 1 odkaz na datový blok, který **místo dat** obsahuje seznam přímých odkazů na datové bloky obsahující vlastní data souboru
- ❑ **DOUBLE INDIRECT** – 1 odkaz 2. nepřímé úrovně
- ❑ **TRIPLE INDIRECT** – 1 odkaz 3. nepřímé úrovně

v linuxových fs (ext\*) ještě FLAGS, počet použitých datových bloků a rezervovaná část – doplňující info (odkaz na rodičovský adresář, ACL, rozšířené atributy)



## 2 základní uspořádání adresáře (!!!)

1. Adresář obsahuje **jméno souboru, atributy, diskovou adresu souboru** (např. adresa 1.bloku)  
(implementuje DOS, Windows)
2. Adresář obsahuje **pouze jméno + odkaz** na jinou datovou strukturu obsahující další informace  
(např. i-uzel) (implementuje UNIX, Linux)

Běžné jsou oba dva způsoby i kombinace

---

## 2 základní uspořádání adresáře (!!)

mail		atributy, diskové adresy
prednasky		atributy, diskové adresy
pracovni		atributy, diskové adresy

a)

mail		
prednasky		
pracovni		

b)

atributy, diskové adresy

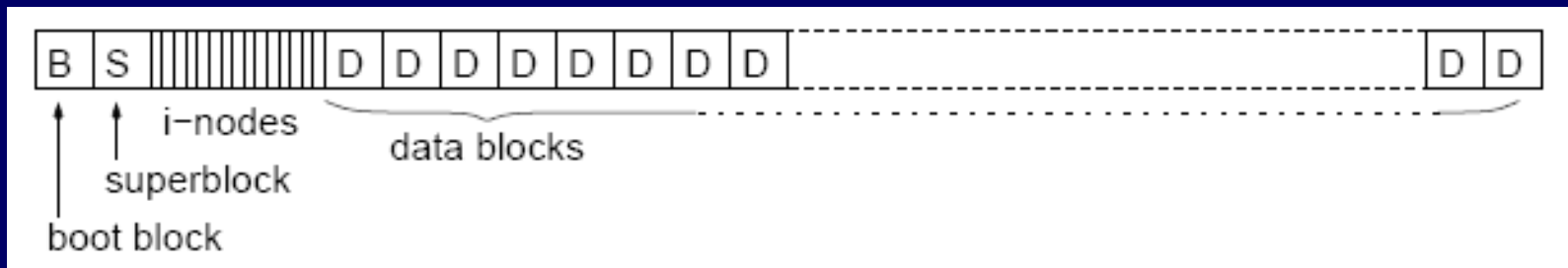
atributy, diskové adresy

atributy, diskové adresy

# Příklad fs (Unix v7)

## □ Struktura fs na disku

- **Boot blok** – může být kód pro zavedení OS
- **Superblok** – informace o fs (počet i-uzlů, datových bloků,...)
- **i-uzly** – tabulka pevné velikosti, číslovány od 1
- **Datové bloky** – všechny soubory a adresáře





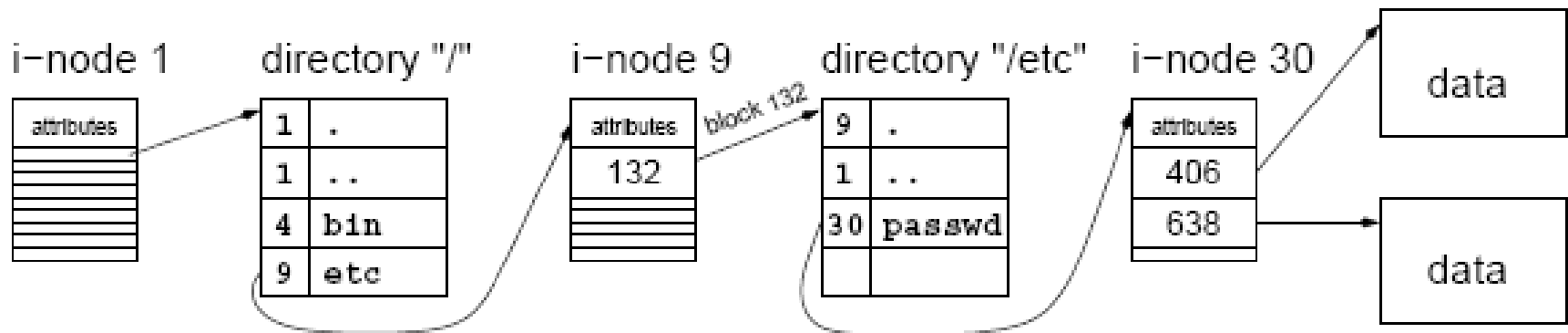
# Implementace souborů – i-uzly

i-uzel obsahuje:

- Atributy
  - Odkaz na prvních 10(až 12) datových bloků souboru
  - Odkaz na blok obsahující odkazy na datové bloky (**nepřímý odkaz**)
  - Odkaz na blok obsahující odkazy na bloky obsahující odkazy na datové bloky (**dvojitě nepřímý odkaz**)
  - **Trojitě nepřímý odkaz**
-

## □ Nalezení cesty k souboru `/etc/passwd`

- V kořenovém adresáři najdeme položku „`etc`“
- i-uzel číslo **9** obsahuje adresy diskových bloků pro adresář `etc`
- V adresáři `etc` (disk blok 132) najdeme položku `passwd`
- i-uzel **30** obsahuje soubor `/etc/passwd`
- (uzel, obsah uzlu, uzel, obsah uzlu)







# Konzistentní fs

Číslo bloku	0	1	2	3	4	5	6	7	8
Výskyt v souborech	1	0	1	0	1	0	2	0	1
Volné bloky	0	1	0	0	1	2	0	1	0

Blok je buď volný, nebo patří nějakému souboru, tj. konzistentní hodnoty v daném sloupci jsou buď (0,1) nebo (1,0)  
Vše ostatní jsou chyby různé závažnosti



# Možné chyby, závažnosti

- (0,0) – blok se nevyskytuje v žádné tabulce
  - Missing blok
  - Není závažné, pouze redukuje kapacitu fs
  - Oprava: vložení do seznamu volných bloků
- (0,2) – blok je dvakrát nebo vícekrát v seznamu volných
  - Problém – blok by mohl být alokován vícekrát !
  - Opravíme seznam volných bloků, aby se vyskytoval pouze jednou



# Možné chyby, závažnosti

- (1,1) – blok patří souboru a zároveň je na seznamu volných
  - Problém, blok by mohl být alokován podruhé !
  - Oprava: blok vyjmemme ze seznamu volných bloků
- (2,0) – blok patří do dvou nebo více souborů
  - Nejzávažnější problém, nejspíš už došlo ke ztrátě dat
  - Snaha o opravu: alokujeme nový blok, problematický blok do něj zkopírujeme a upravíme i-uzel druhého souboru
  - Uživatel by měl být informován o problému



# Jak funguje žurnál (!!)

1. Zapiši do žurnálu
2. Když je žurnál kompletní, zapišeme značku ZURNAL\_KOMPLETNI
3. Začneme zapisovat datové bloky
4. Je-li hotovo, smažeme žurnál



# Žurnál – ošetření výpadku (!!)

- Dojde-li k výpadku elektřiny → nebyl korektně odmontovaný oddíl se souborovým systémem → pozná
- Podívá se do žurnálu:
  - a) **Je prázdný**  
→ není třeba nic dělat
  - b) **Je tam nějaký zápis, ale není značka ZURNAL\_KOMPLETNI**  
-> jen smažeme žurnál
  - c) **V žurnálu je zápis včetně značky ZURNAL\_KOMPLETNI**  
-> přepíšeme obsah žurnálu do datových bloků



# Co žurnálovat?

- Všechny zápisy, tj. i do souborů
    - Zapisují se metadata i data
    - pomalejší
  
  - Zápisy metadat
    - Rychlejší
    - Může dojít ke ztrátě obsahu souboru, ale nerozpadne se struktura adresářů
-

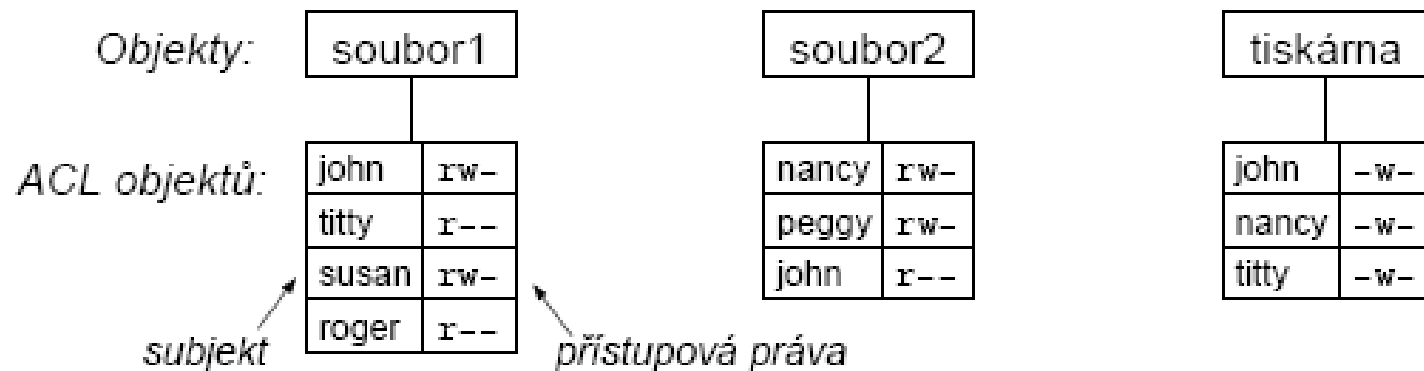


# ACL x capability list

- Dvě různé podoby
  - **ACL** – **s objektem je sdružen** seznam subjektů a jejich přístupových práv
  - **Capability list** [kejpa-] – **se subjektem je sdružen** seznam objektů a přístupových práv k nim
-

# ACL (Access Control Lists)

- S objektem je sdružen seznam subjektů, které mohou k objektu přistupovat
- Pro každý uvedený subjekt je v ACL množina přístupových práv k objektu







# ACL – příklad (!)

Např v **NTFS**:

Se souborem **data1.txt** je spojena následující **ACL tabulka**, která určuje, kdo smí co s daným souborem dělat. Počet řádek tabulky záleží na tom, pro kolik uživatelů skupin budeme práva nastavovat.

Klasická unixová práva (u,g,o) jsou příliš limitovaná – když chceme více skupin, více uživatelů atd. potřebujeme ACL.

uživatel / skupina	id uživatele	práva
0	505 (Pepa)	rw
1	101 (Studenti)	r
1	102 (Zamestnanci)	rw

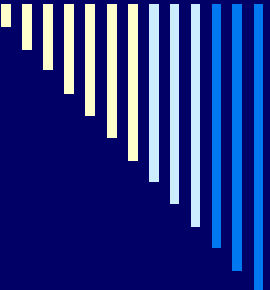


# ACL příkazy

- **getfacl** soubor
- **setfacl** -m user:pepa:rw s1.txt

Přečtěte si článek:

<http://www.abclinuxu.cz/clanky/bezpecnost/acl-prakticky>



# Mechanismus capability lists (C-seznamy)


- S každým subjektem (procesem) sdružen seznam objektů, kterým může přistupovat a jakým způsobem (tj. přístupová práva)
  - Seznam se nazývá capability list (C-list)
  - Jednotlivé položky - capabilities
-



# Capability

- Problém – zjištění, kdo všechno má k objektu přístup
- Zrušení přístupu velmi obtížné – najít pro objekt všechny capability + odejmout práva
- Řešení: odkaz neukazuje na objekt, ale na nepřímý objekt  
systém může zrušit nepřímý objekt, tím zneplatní odkazy na objekt ze všech C-seznamů

# Typy záloh



manipulací s  
atributech  
archive

## □ normální

- zálohuje, označí soubory jako "zazálohované"

## □ copy

- zálohuje, ale neoznačí jako "zazálohované"

## □ incremental

- zálohuje pouze vybrané soubory, tj. pokud nebyly "zazálohované" nebo byli změněny a označí je jako "zazálohované"

## □ differential

- viz předchozí, ale neoznačuje jako "zazálohované"
- změny, které proběhly od plné zálohy
- diferenciální zálohy nejsou na sobě závislé

## □ daily

- zálohuje soubory změněné dnes, ale neoznačuje je jako "zazálohované"



# Co se děje při spuštění PC?

01. Pustíme proud

02. **Power on self-test** (řízen BIOSem)

test operační paměti, grafické karty, procesoru

test pevných disků, dalších ATA/SATA/USB zařízení

03. spustí z ROM paměti BIOSu **bootstrap loader**

prohledá boot sektor bootovacího zařízení (dle CMOS)

boot sektor - první sektor na disku, je zde umístěn zavaděč systému (boot loader)

[http://cs.wikipedia.org/wiki/Power\\_On\\_Self\\_Test](http://cs.wikipedia.org/wiki/Power_On_Self_Test)



# Co se děje při spuštění PC?

04. pustí se **zavaděč** (GRUB, GRUB2, LILO, ...)

může se skládat z více stupňů (stage), v boot sektoru je stage1  
možnost zvolit si jaký systém nabootuje (Linux, Windows)

05. zavaděč nahraje **jádro do paměti** a spustí ho

jádro píše na obrazovku info zprávy

- můžeme prozkoumat příkazem **dmesg**

06. první proces **init**

/sbin/init , načte /etc/inittab , spouštění a vypínání služeb

/etc/rc.d/rcX.d

---

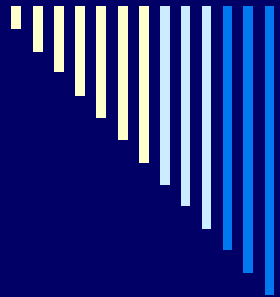


# Co se děje při spuštění PC?

07. spustí program **getty** na virtuálních terminálech  
zadáme uživatelské jméno

08. spustí se **login**  
vyžádá si heslo, zkontroluje v `/etc/passwd`, `/etc/shadow` či jinde





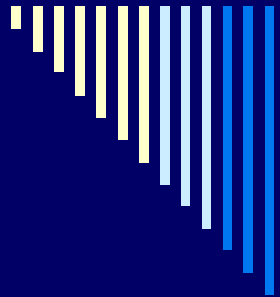
# Komponenty Linuxu

## □ Monolitické jádro OS

- Jediné běží v režimu jádra
- Poskytuje základní abstrakce (procesy, virtuální paměť, soubory)
- Aplikace žádají o služby jádra prostřednictvím **volání služeb** systému
- Umožňuje za běhu přidávat a vyjímat **moduly** (ovladače)
  - Přidat / zrušit části kódu jádra na žádost

## □ Systémové knihovny

- Standardní funkce pro použití v aplikacích
- Některé komunikují s jádrem OS (write apod.)
- Mnoho knihovných funkcí jádro nevolá (sin, cos, tan, ..)



# Přístup do paměti - Linux

- mechanismus **stránkování**
  - program přistupuje na virtuální adresu
  - využití tabulky stránek pro převod na fyzickou adresu
  - mapování není – výjimka
  - v tabulce stránek – práva přístupu a druh přístupu
  - cache na několik posledních mapování – TLB (transaction lookaside buffer)