

03. Synchronizace procesů

ZOS 2006, L. Pešička



Administrativa ☺

- 1. zápočtový test
 - 7.11.2006 (út), EP130, 18:30

 - praktická cvičení před testem
 - (slide upraven na aktuální termín)
-



Plánování procesů

- **Krátkodobé – CPU scheduling**
kterému z připravených procesů bude přidělen procesor; vždy ve víceúlohovém
 - **Střednědobé – swap out**
odsun procesu z vnitřní paměti na disk
 - **Dlouhodobé – job scheduling**
výběr, která úloha bude spuštěna
dávkové zpracování
(dostatek zdrojů – spust' proces)
 - **Liší se – frekvence spouštění**
-



Plánování procesů

- **Stupeň multiprogramování**
 - Počet procesů v paměti
 - Ovlivňuje long term scheduler
 - Snižuje middle term scheduler
-

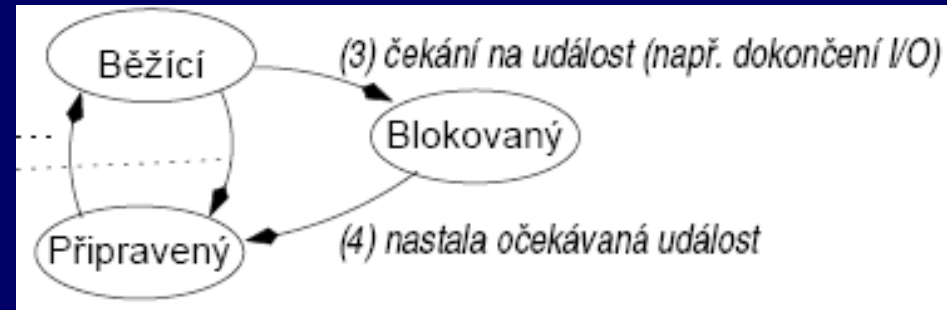
Plánování

□ Nepreemptivní

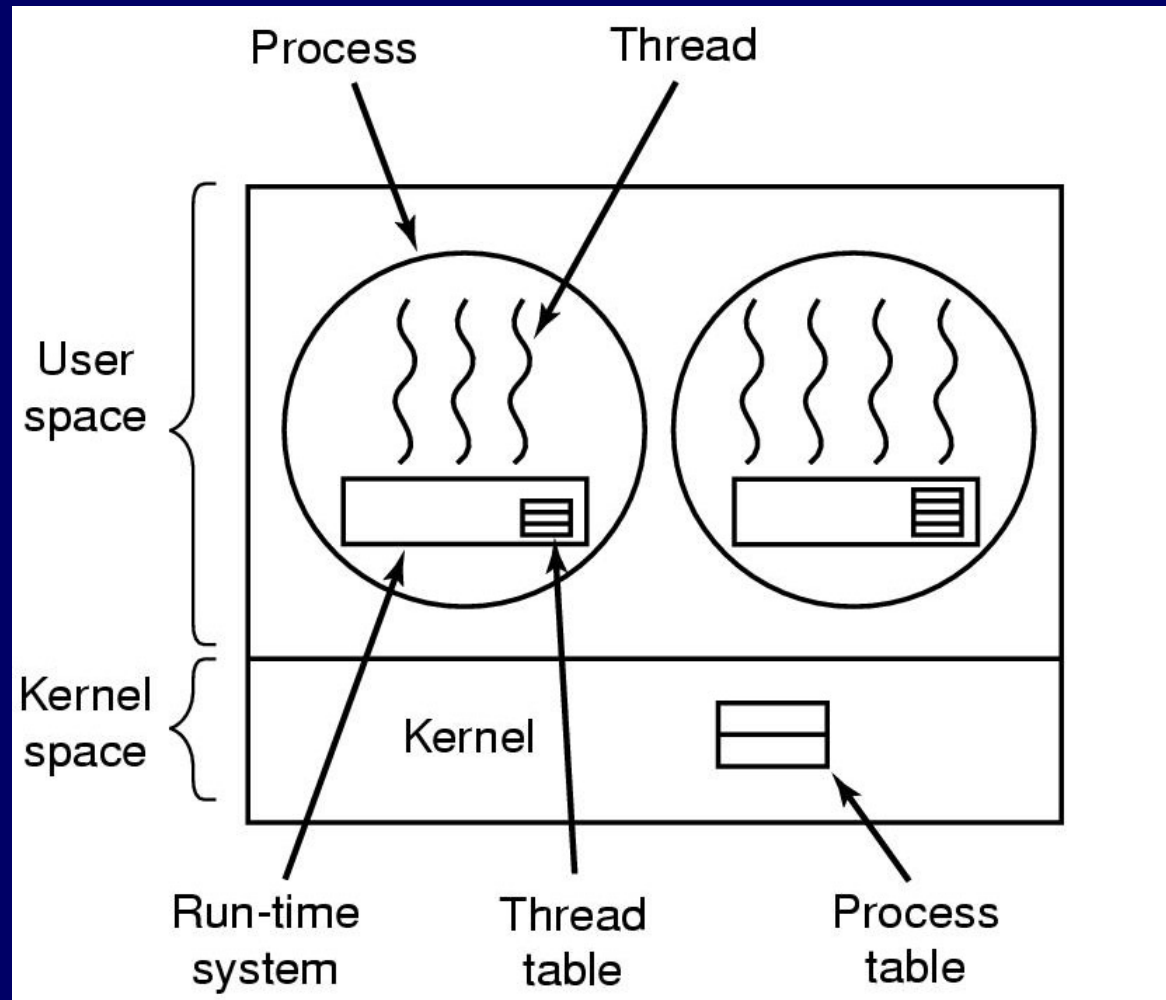
- Proces skončí
- Běžící -> Blokováný
 - Čekání na I/O operaci
 - Semafor
 - Čekání na ukončení potomka

□ Preemptivní

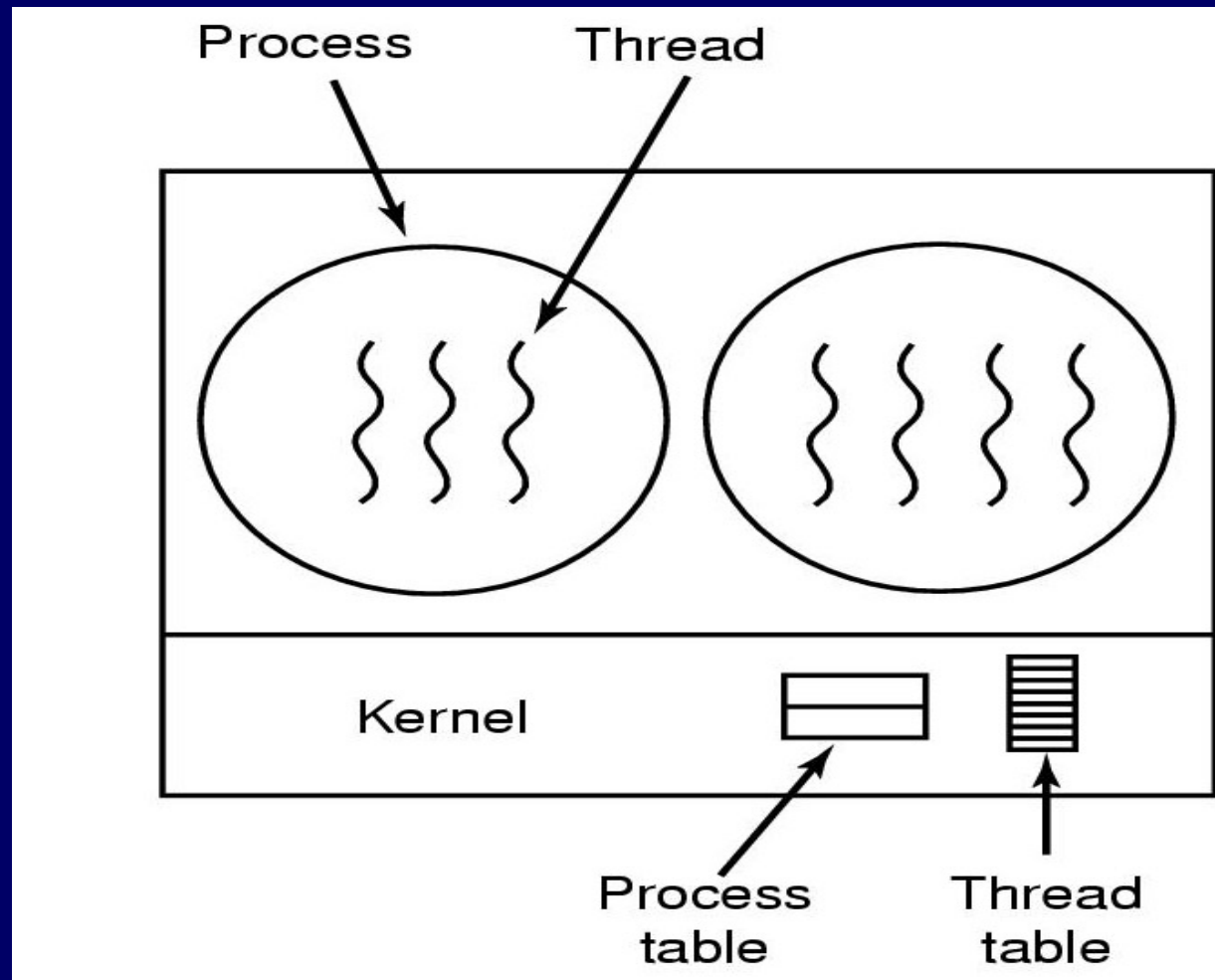
- Běžící -> Připravený
 - Uplynulo časové kvantum



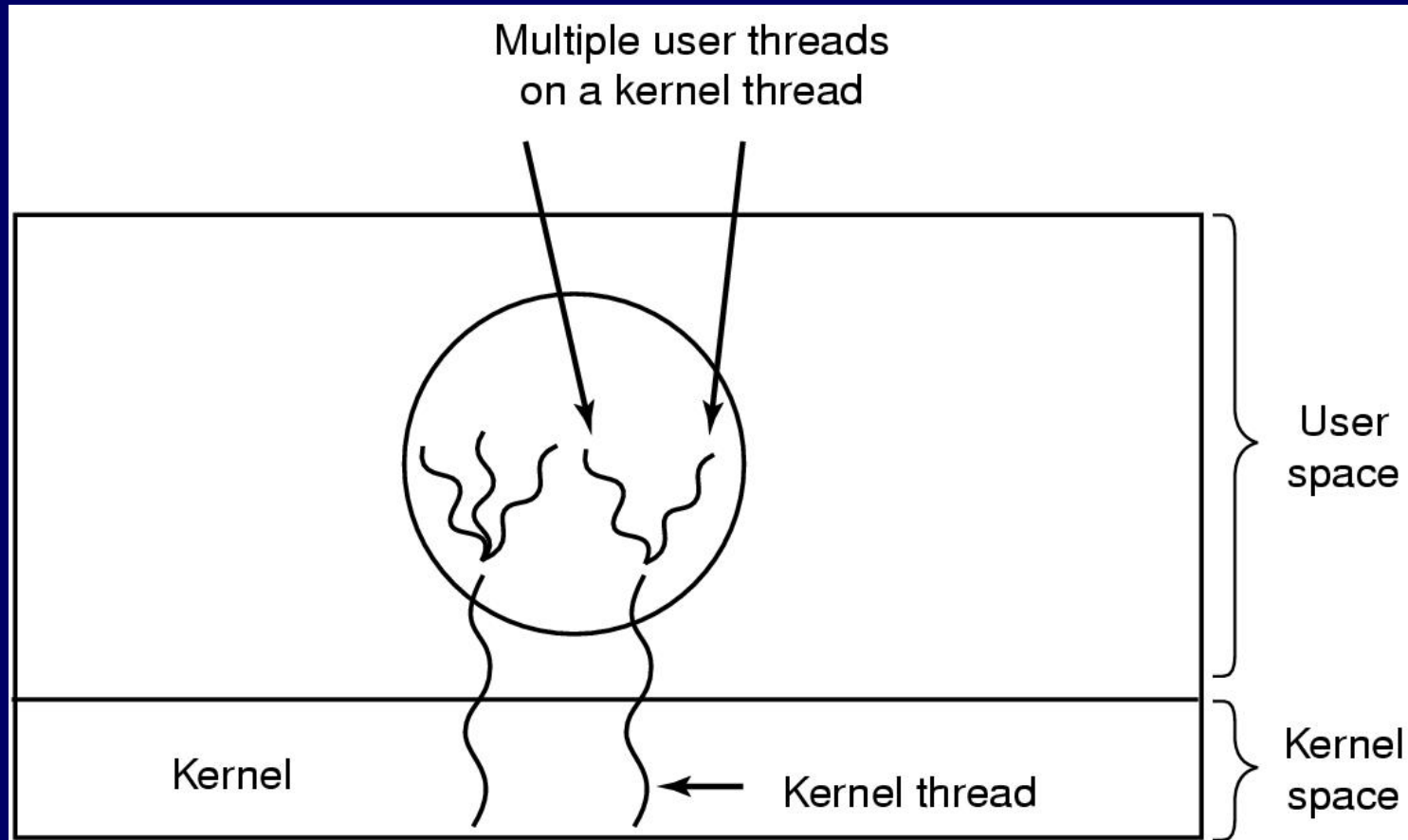
Vlákna v User Space



Vlákna v jádře



Hybridní implementace





Modely - vlákna

□ one-to-one

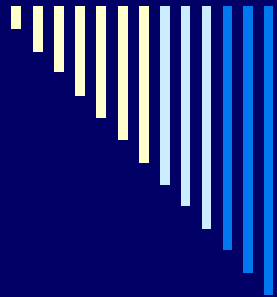
- Každé vlákno – separátní proces v jádře
- Plánovač jádra je plánuje jako běžné procesy

□ many-to-one

- User level plánovač vláken
- Z pohledu jádra – pouze 1 proces

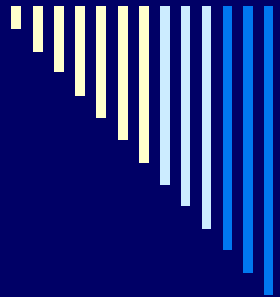
□ many-to-many

- Komerční unixy (Solaris, Digital Unix, IRIX)
-



Linux

- Systémové volání **clone()**
 - Zobecněná verze **fork()**
 - One-to-one model
 - Dovoluje novému procesu sdílet s rodičem
 - Paměťový prostor
 - File descriptors
 - Signal handlers
 - Specifické pro Linux, není přenositelné (portable)



Pthreads

- Historicky každý výrobce svoje řešení
- UNIX – IEEE POSIX 1003.1c standard (1995)
 - Implementace POSIX threads – Pthreads

- gcc -lpthread ahoj.c
- <http://yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html>

- Příklad - eryx

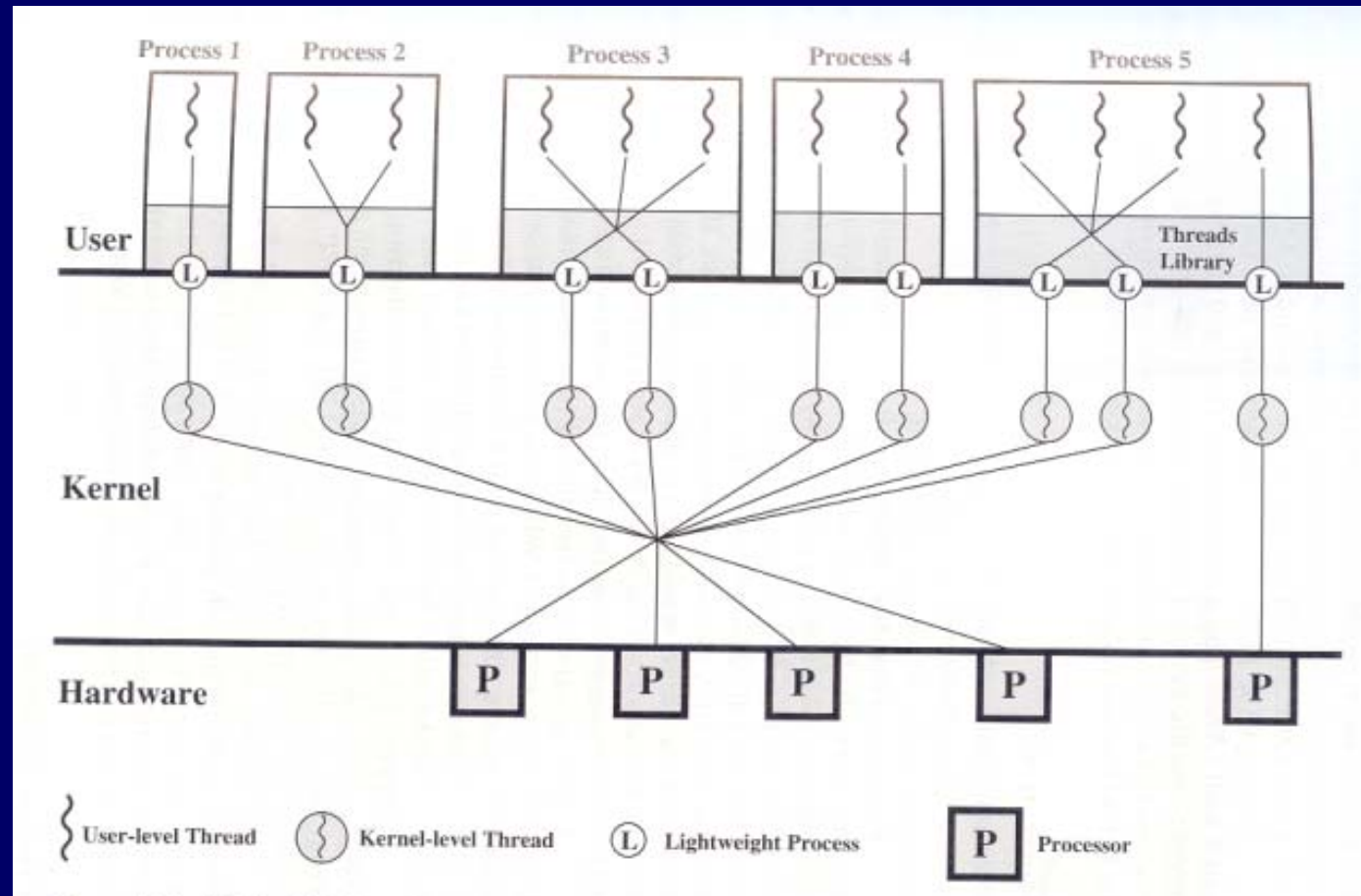


Vlákna - Solaris

- Uživatelská vlákna, vlákna jádra
 - **Lehké procesy (LWP)**
 - Každý proces – min. 1 LWP
 - Uživatelská vlákna multiplexována mezi LWP procesy
 - Pouze vlákno napojené na některý LWP může pracovat
 - Ostatní blokována nebo čekají na uvolnění LWP

 - Každý LWP proces – jedno vlákno jádra
 - Další vlákna jádra bez LWP – např. obsluha disku
 - Vlákna jádra – multiplexována mezi procesory
-

Vlákna Solaris





Vlákna - Java

- Vlákno – instance třídy `java.lang.Thread`
- Odvodit potomka, překrýt metodu `run()`
 - Vlastní kód vlákna
- Spuštění vlákna – volání metody `start()` třídy `Thread`
- Třída implementující rozhraní `Runnable`

```
class Něco implements Runnable {  
    public void run() { ... } }
```



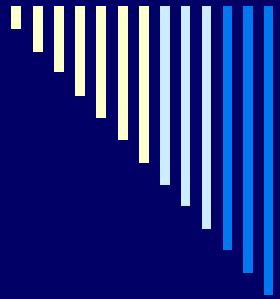
Synchronizace procesů

- Časový souběh
 - Kritická sekce
 - Algoritmy pro přístup do kritické sekce
 - Semafory
-



Časový souběh

- Procesy sdílejí společnou paměť – čtení a zápis
- Může nastat **časový souběh** (**race condition**)
- Dva procesy zvětšují asynchronně společnou proměnnou X



Příklad dvou procesů

```
cobegin
```

```
...
```

```
x := x + 1;
```

```
...
```

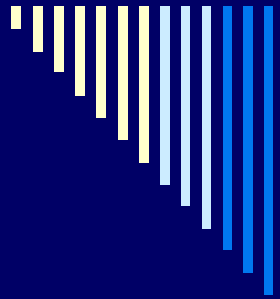
```
||
```

```
...
```

```
x := x + 1;
```

```
...
```

```
coend
```



Příkaz na nízkourovňové inst.

- $x := x + 1;$
- Načtení hodnoty x do registru (LD R, x)
- Zvýšení hodnoty x (INC R)
- Zápis nové hodnoty do paměti (LD x, R)
- Pokud oba procesy provedou příkazy **sekvenčně**, bud mít x správně $x+2$



Chybné pořadí vykonání

❑ Přepnutí v nevhodném okamžiku.. Pseudopar.

❑ P1: LD R, x // x je 0, R je 0

❑ P2: LD R, x // x je 0, R je 0

❑ INC R // x je 0, R je 1

❑ LD x, R // x je 1, R je 1

❑ P1: // x je 1, R je 0 – rozpor

❑ INC R // x je 1, R je 1

❑ LD x, R // x je 1, R je 1

❑ Výsledek – **chyba**, neprovedlo se každé zvětšení



Chybné vykonání – 2 CPU

- Chyba i při paralelním běhu

Proces 1:

```
LD R, x
INC R
LD x, R
...
```

Proces 2:

```
...
LD R, x
INC R
LD x, R
```



Př. bankovní transakce

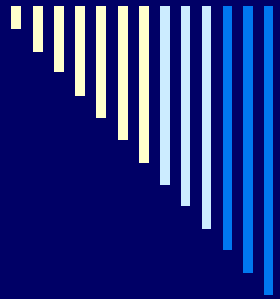
- Dva procesy přístup do databáze
- $\text{Účet} := \text{účet} + 20\ 000$ 1. proces
- $\text{Účet} := \text{účet} - 15\ 000$ 2. proces
- Správný výsledek?
- Možné výsledky?



Časový souběh – kdy nastane

- Přidávání prvku do seznamu

- Přístup do souboru
 - 2 procesy chtějí vytvořit soubor a zapsat do něj
 - 1. proces – zjistí, že soubor není
 - ... přeplánování ...
 - 2. proces – zjistí, že soubor není, vytvoří a zapíše
 - 1. proces – pokračuje, vytvoří a zapíše
 - znehodnotí činnost druhého procesu



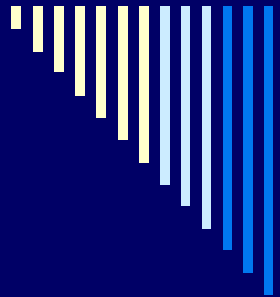
Výskyt souběhu

- časový souběh se projevuje **nedeterministicky**
- většinu času běží programy bez problémů
- hledání 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**

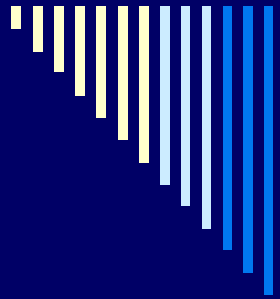


Kritická sekce

- 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
-

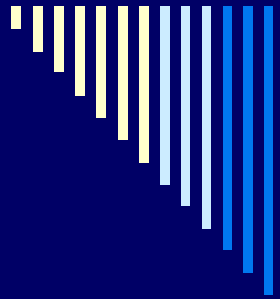


Společná datová oblast

- hlavní paměť

 - soubor
 - pokud 1 proces pracuje s jinou hodnotou, než jakou očekává jiný proces
 - **zamykání částí souboru** – řeší časový souběh

 - každá **kritická sekce** se vztahuje ke **konkrétním datům**, ke kterým se v ní přistupuje
-



Struktura procesů

```
cobegin
```

```
P1: while true do           // nekonečná smyčka
```

```
  begin
```

```
    nevinná_činnost;      // pouze s vlastními daty
```

```
    kritická_sekce         // přístup do sdílených dat
```

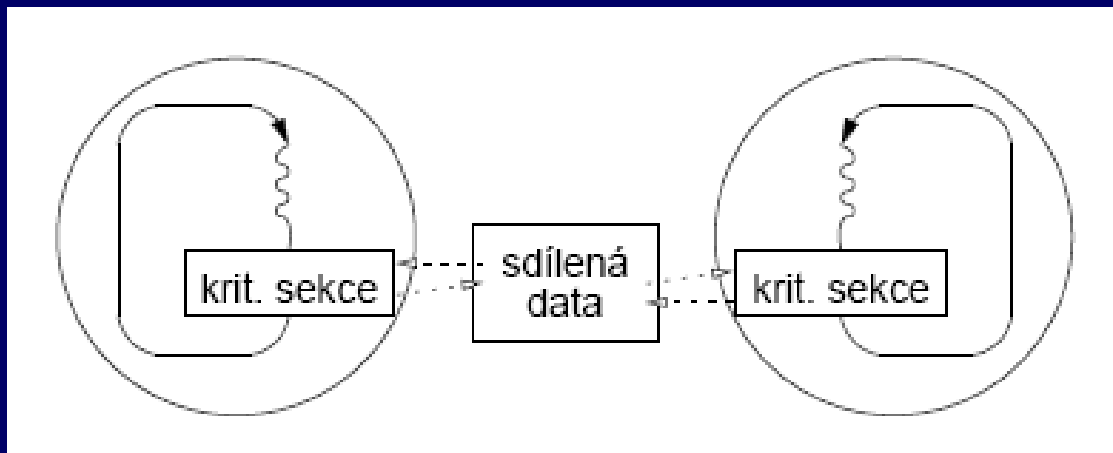
```
  end
```

```
  ||
```

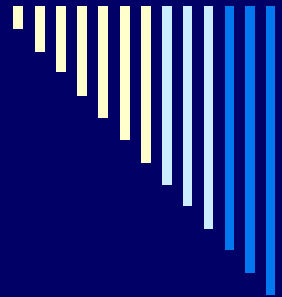
```
P2: ...                   // totéž co P1
```

```
coend
```

Kritická sekce



Proces, který chce do kritické sekce musí počkat, až z ní jiný proces vystoupí



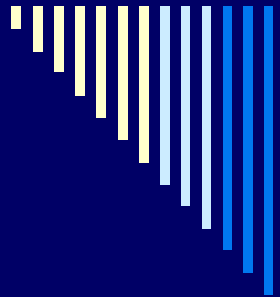
Řešení časového souběhu - pravidla

- **Vzájemné vyloučení** - žádné dva procesy nesmějí být **současně** uvnitř své kritické sekce
- Proces běžící **mimo kritickou sekci nesmí blokovat** jiné procesy (např. jim **bránit ve vstupu** do kritické sekce)
- 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

□ přeplánování procesů

- pouze jako **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
- není dovoleno v **uživatelském režimu** (jinak pokud uživatel zakáže a nepovolí...)
- používáno často **uvnitř jádra** OS
- ale **není** vhodné pro **uživatelské procesy**



Aktivní čekání - předpoklady

- zápis a čtení ze společné datové oblasti jsou **nedělitelné** operace
 - současný přístup více procesů ke stejné oblasti povede k sekvenčním odkazům v neznámém pořadí
 - platí pro data \leq délce slova
 - kritické sekce nemohou mít přiřazeny **prioritu**
 - relativní **rychlost** procesů je **neznámá**
 - proces se může **pozastavit** mimo kritickou sekci
-



Algoritmus 1

– procesy přistupují střídavě

```
program striktni_stridani;  
var turn: integer;  
begin  
    turn := 1;  
    cobegin  
        P1: while true do  
            begin  
                while turn = 2 do; // čekací smyčka  
                    KS1; // kritická sekce  
                turn := 2 // a může druhý  
            end  
    end
```



Algoritmus 1 pokračování

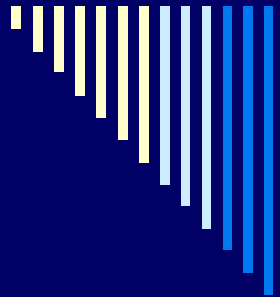
||

```
P2: while true do
  begin
    while turn = 1 do; // čekací smyčka
      KS2; // kritická sekce
    turn:= 1 // a může první
  end
coend
end
```



Algoritmus 1

- Problém – porušuje pravidlo 2
- Pokud je jeden proces podstatně **rychlejší**, **nemůže** vstoupit do kritické sekce 2x za sebou



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**

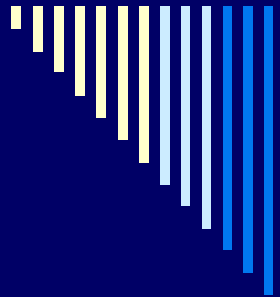
- **spin lock**



Algoritmus - Peterson

- První úplné řešení navrhl Dekker, ale je poměrně složité

- Jednodušší a elegantnější algoritmus navrhl Peterson (1981)
 - viz dále řešení pro 2 procesy
 - lze i zobecnit



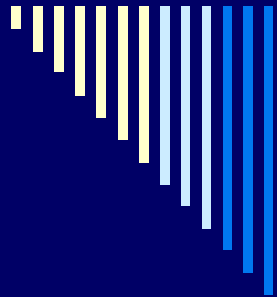
Peterson – enter_CS()

```
program petersonovo_reseni;  
var turn: integer;  
interested: array [0..1] of boolean;      // na začátku {false, false}  
  
procedure enter_CS(process: integer);  
var other: integer;  
begin  
    other:=1-process;                    // ten druhý proces  
    interested[process]:=true;           // oznámí zájem o vstup  
    turn:=process;                       // nastaví příznak  
    while turn=process and interested[other]=true do ;  
end;
```



Peterson – leave_CS()

```
procedure leave_CS(process: integer);  
  begin  
    interested[process]:=false;      // oznámí odchod z KS  
  end;
```



Peterson

```
begin
  interested[0]:=false; // inicializace
  interested[1]:=false;
  cobegin
    while true do {cyklus - vlákno 1}
      begin
        enter_CS(0);
        KS1;
        leave_CS(0);
      end {while}
    || {vlákno 2 analogické}
  coend
end.
```



Peterson - vysvětlení

- na začátku není v KS žádný proces
- **první proces volá enter_CS(0)**
 - interested[0] := true; turn := 0;
 - nebude čekat ve smyčce, interested[1] je false
- **nyní proces 2 volá enter_CS(1)**
 - interested[1] := true; turn := 1;
 - čeká ve smyčce, dokud interested[0] nebude false (leave_CS)
- **pokud oba volají enter_CS téměř současně...**
 - oba nastaví **interested** na **true**
 - oba nastaví **turn** na své číslo **ALE provede se sekvenčně, 0 OR 1**
 - např. druhý proces bude jako druhý ☺, tedy turn bude 1
 - oba se dostanou do while, první proces projde, druhý čeká



Spin lock s instrukcí TSL

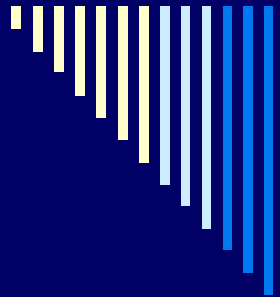
- hw podpora
- většina počítačů – instrukci, která **otestuje hodnotu a nastaví paměťové místo v jedné nedělitelné operaci**

- operace **Test and Set Lock** – TSL, TS:
- **TSL R, lock**
 - R je registr CPU
 - lock – buňka paměti, 0 false nebo 1 true; boolean; provádí:
 - LD R, lock
 - LD lock, 1



TSL

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



TSL - použití

- Proměnná typu zámeček – na počátku 0
- Proces, který chce vstoupit do KS – test
 - Pokud 0, nastaví na 1 a vstoupí do KS
 - Pokud 1, čeká
- Pokud by TSL nebyla atomická
 - Jeden proces přečte, vidí 0 .. Přeplánování..
 - Druhý proces přečte, vidí 0, nastaví 1, vstoupí KS
 - První proces naplánován, zapíše 1 a je také v KS



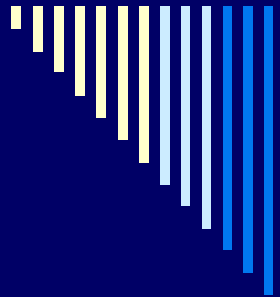
Implementace zámku

Spin_lock:

```
TSL R, lock      ;; atomicky R=lock, lock=1
CMP R, 0         ;; byla v lock 0?
JNE spin_lock    ;; pokud ne cykluj dál
RET              ;; návrat, vstup do KS
```

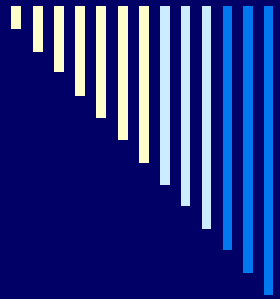
Spin_unlock:

```
LD lock, 0      ;; ulož hodnotu 0 do lock
RET
```



Implementace zámku – pozn.

- Cyklus v `spin_lock` dokud `lock` je 1
- Když někdo vyvolá `spin_unlock`, přečte 0 a může vstoupit do KS
- Pokud na vstup do KS čeká více procesů
 - Hodnotu 0 přečte jenom jeden z nich (první kdo vykoná TSL)



Implementace – jádro Linuxu

spin_lock:

TSL R, lock

CMP R, 0 ;; byla v lock 0 ?

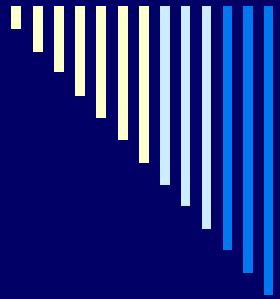
JE cont ;; pokud byla, skočíme

Loop: **CMP lock, 0** ;; je lock 0 ?

JNE loop ;; pokud ne, skočíme

JMP spin_lock ;; pokud ano, skočíme

Cont: RET ;; návrat, vstup do KS



Náhrada TSL

□ Uniprocessor

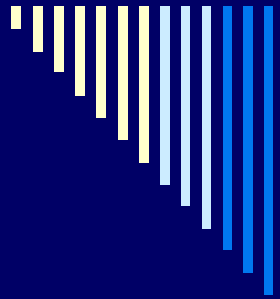
- Nedělitelnost zakázáním přerušení (DI/EI, CLI/STI)

□ Multiprocessor

- Primitivní operace s uzamčením sběrnice

□ Př. I8086:

- MOV AL, 1 ; do AL 1
- LOCK XCHG AL, X ; zamkne sběrnici pro XCHG
; zamění AL a X



TSL – jak v BACI

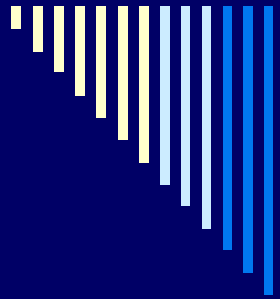
```
atomic function TSL (var x: boolean) : boolean;
```

```
begin
```

```
    TSL := x;
```

```
    x := true;
```

```
end;
```



Implementace spin-locku

```
type lock = boolean;
```

```
procedure spin_lock (var m: lock);
```

```
begin
```

```
    while TSL(m) do ;    {čeká dokud je m true}
```

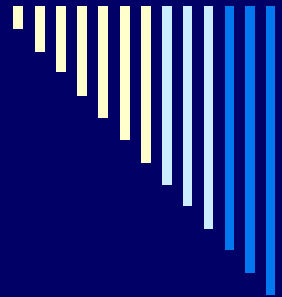
```
end;
```



Implementace spin-locku

```
procedure spin_unlock (var m: lock);  
begin  
    m := false;  
end;
```

Pozn. V literatuře TSL někdy se nastavuje true,
někdy false



Problém řešení s aktivním čekáním

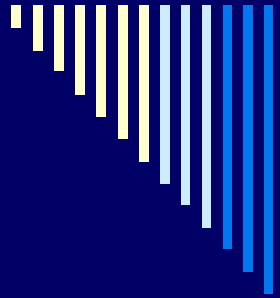
- Peterson, spin-lock
- Ztracený čas CPU
 - Jeden proces v KS, další může ve smyčce přistupovat ke společným proměnným
 - krade paměťové cykly aktivnímu procesu
- Problém inverze priorit
 - Pouze zde připustíme, že procesy mají prioritu
 - Dva procesy, jeden s vysokou H a druhý s nízkou L prioritou, H se spustí jakmile připraven



Problémy akt. čekání

- L je v **kritické** sekci
 - H se stane připravený (např. má vstup)
 - H začne **aktivní čekání**
 - L ale **nebude** už nikdy **naplánován**, nemá šanci dokončit KS
 - H bude aktivně **čekat** do **nekonečna**

 - **Problém inverze priorit**
-



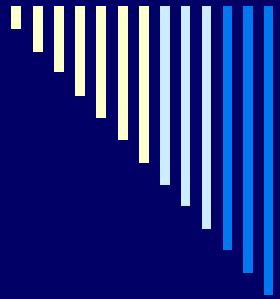
Řešení

- hledala se primitiva, která proces zablokuje, místo aby čekal aktivně
-



Semaforey

- Dijkstra (1965) navrhl primitivum, které zjednodušuje **komunikaci** a **synchronizaci** procesů – **semaforey**
- **Semafor** – proměnná, obsahuje nezáporné celé číslo
- Semaforu lze **přiřadit hodnotu pouze** při deklaraci
- Nad semaforey **pouze operace** $P(s)$ a $V(s)$



Operace P

- **Operace P(S):**

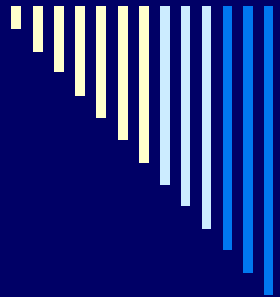
- Pokud $S > 0$, sníží S o 1

- Jinak pozastaví proces, který chtěl provést operaci P



Operace V

- **Operace V(S):**
- Pokud je nad semaforem S **zablokovaný** jeden nebo více procesů, **vzbudí** jeden z procesů; proces pro vzbuzení je vybrán **náhodně**
- Jinak zvýší S o 1



Poznámky

- Operace P a V jsou **nedělitelné** (atomické) akce
 - Jakmile **začne** operace nad semaforem – nikdo k němu **nemůže přistoupit** dokud operace neskončí nebo se nezablokuje

- Několik procesů **současně** ke stejnému semaforu
 - Operace se provede **sekvenčně** v **libovolném** pořadí

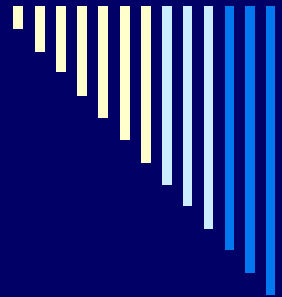


Poznámky - terminologie

- V literatuře $P(s)$ někdy $wait(s)$ nebo $down(s)$
 - $V(s)$ nazýváno $signal(s)$ nebo $up(s)$

 - Původní označení z holandštiny
 - P proberen – otestovat
 - V verhogen – zvětšit

 - *Pomůcka – např. abecední pořadí operací*
-



Vzájemné vyloučení – pomocí semaforů

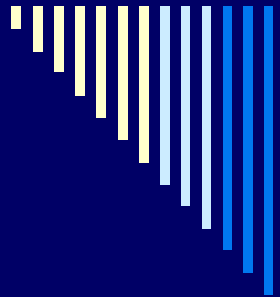
- Vytvořit semafor s hodnotou 1
- Před **vstupem** do KS – $P(s)$
- Po **vystoupení** z KS – $V(s)$

- Je-li libovolný proces v **KS**
 - Potom S je **0**, jinak S je 1



Vzájemné vyloučení

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



Použití semaforů

- **Vzájemné vyloučení**
 - **Mutexy**, binární semaforey .. 0 a 1

- **Kooperace procesů**
 - Problém omezených zdrojů
 - Obecné semaforey .. 0 1 2 ..



Vzájemné vyloučení - KS

- Procesy **soupeří** o zdroj
 - Ke zdroji může přistupovat **víc než 1** proces v daném **čase**
 - Každý proces může **existovat bez ostatních**
 - Interakce **POUZE** pro zajištění **serializace** přístupu ke zdroji
-



Kooperace procesů

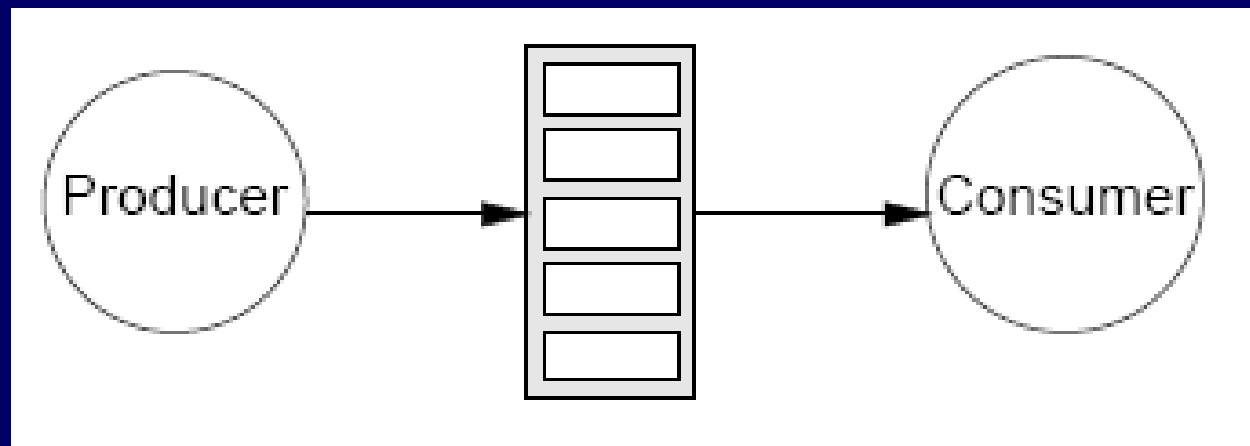
- Procesy se navzájem **potřebují**, potřeba vzájemné **výměny** informací
 - Nejjednodušší případ – pouze **synchronizační** signály
 - Obvykle – i **další informace** – např. zasíláním zpráv
-



Problém producent-konzument

- Problém ohraničené vyrovnávací paměti (bounded buffer problem, Dijkstra 1968)
- Dva procesy **společnou paměť** (buffer) **pevné velikosti N** položek
- Jeden proces – **producent** **generuje** nové položky a ukládá do vyrovnávací paměti
- Paralelně **konzument** – data vyjímá a **spotřebovává**

Producent - konzument



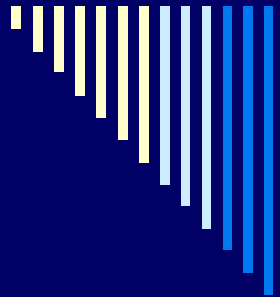
Př. **Hlavní program** tiskne x **tiskový server**, **blok** – 1 stránka

Př. **Obslužný prog.** **čte data ze zařízení** x hlavní program je **zpracovává**



Různé rychlosti procesů

- Procesy – různé rychlosti – zabezpečit, aby nedošlo k přetečení / podtečení
- **Konzument** musí být schopen **čekat** na producenta, **nejsou-li data**
- **Producent** – **čekat** na konzumenta, je-li **buffer plný**



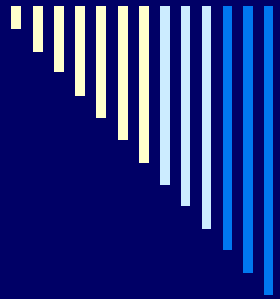
Prod-konz. pomocí semaforů

- Pro **synchronizaci** obou procesů
- Pro **vzájemné vyloučení** nad KS
- Proces se zablokuje **P**, jiný ho vzbudí **V**
- Semaforey:
 - **e** – počet **prázdných** položek v bufferu dostupných producentovi (empty)
 - **f** – počet **plných** položek ještě nespotřebovaných konz. (full)



Třetí semafor

- Semafor **m** pro **vzájemné vyloučení**
- Přidávání a vybírání ze společné paměti **může být** obecně **kritickou sekci**



P&K - implementace

Var

```
e: semaphore = N;           // prázdných  
f: semaphore = 0;          // plných  
m: semaphore = 1;          // mutex
```



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}
```



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
```

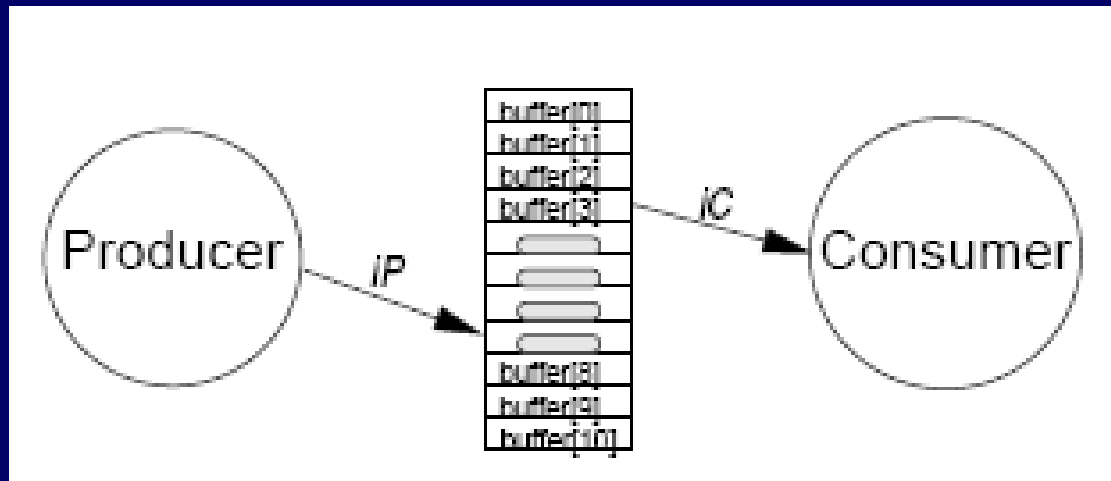
```
    zpracuj záznam;
```

```
end {while}
```

```
coend.
```

P&K poznámky

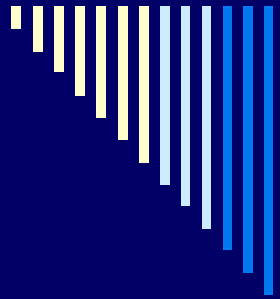
- Vyrovnávací paměť se často implementuje jako **pole** – buffer [0..N-1]





P&K poznámky

- Oba procesy – vlastní index do pole buffer
- Např. operace přidej do bufferu:
□ $buffer[iP]:=polozka; iP:=(iP+1) \bmod N;$
- Pokud je buffer jako pole, vzájemné vyloučení pro přístup dvou procesů nebude potřebné, každý přistupuje pouze k těm, ke kterým má přístup dovolen operací V(s)



Literatura

obrázek Solaris LWP procesy

z knížky

W.Stalling: Operating systems –
Internals and design Principles
