

03. Synchronizace procesů

ZOS 2016, L. PEŠIČKA

Opakování

Kde je uložený PID procesu?

v PCB v tabulce procesů

Kde leží tabulka procesů?

v RAM

Jaké systémové volání vytvoří nový proces?

Linux: `fork()` Windows: `CreateProcess()`

Jakým způsobem spustím jiný program?

Linux: `execve()` , často v kombinaci s `fork()`

Windows: v `CreateProcess` řekneme, jaký program spustit

Poznámky - execve

Spuštění jiného programu v rámci aktuálního procesu

- ❑ nový programový kód
- ❑ nový zásobník
- ❑ jedno vlákno
- ❑ odmapuje původně sdílenou paměť
- ❑ zavře původní fronty zpráv
- ❑ zavře pojmenované semaforey
- ❑ ... a další činnosti viz **man execve**
- ❑ PID zůstává

Stavy procesů

– poznámky k implementaci v Linuxu

Zombie (defunct)

- Proces dokončil svůj kód
- Stále má záznam v tabulce procesů
- Čekání, dokud rodič nepřechte exit status (voláním `wait()`); příkaz `ps` zobrazuje stav “Z”
- Zombie zabírá PID a místo v paměti pro deskriptor procesu

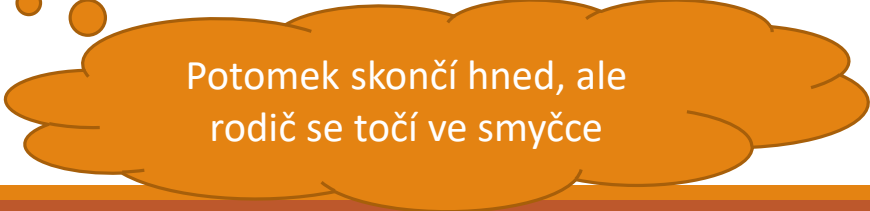
Sirotek

- Jeho kód stále běží, ale skončil rodičovský proces
- Adoptován procesem `init`

Jak na zombii?

```
#include <stdio.h>

int main (void) {
    int i,j;
    i = fork();
    if (i == 0)
        printf ("Jsem potomek s pidem %d, rodic ma %d\n", getpid(), getppid());
    else {
        printf ("Jsem rodic s pidem %d, potomek ma %d\n", getpid(), i);
        for (j=10; j<100; j++) j=11; // rodic neskonci, nekonečná smyčka
    }
}
```




Potomek skončí hned, ale
rodič se točí ve smyčce

Plánování procesů

Krátkodobé – CPU scheduling

kterému z připravených procesů bude přidělen procesor;
je vždy ve více úlohovém



typický plánovač jak
jej známe

Střednědobé – swap out

odsun procesu z vnitřní paměti na disk

Dlouhodobé – job scheduling

výběr, která úloha bude spuštěna
dávkové zpracování (dostatek zdrojů – spustí proces)

Liší se – frekvencí spouštění plánovače

Plánování procesů

Stupeň multiprogramování

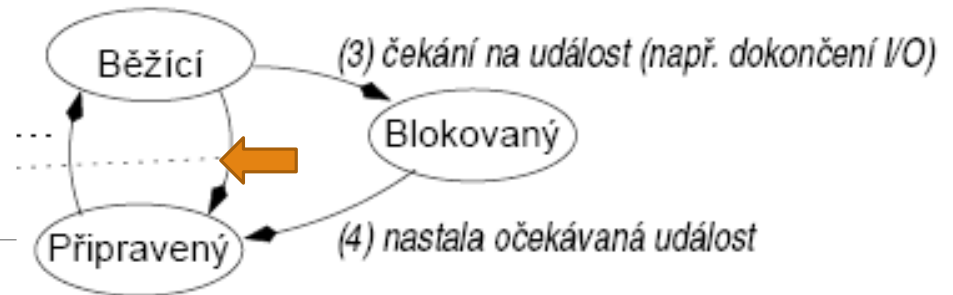
- Počet procesů v paměti
- Zvyšuje jej: long term scheduler (dlouhodobý plánovač)
- Snižuje jej: middle term scheduler (střednědobý plánovač)

Ne v každém OS musí být všechny
tři typy plánovače, typicky jen
krátkodobý plánovač

Plánování

Nepreemptivní

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



Proces opustí CPU:
jen když skončí, nebo
se zablokuje

Preemptivní

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

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

Vlákna

Vlákna mohou být implementována:

- V jádře
- V uživatelském prostoru
- Kombinace

Zná jádro pojem vlákna?

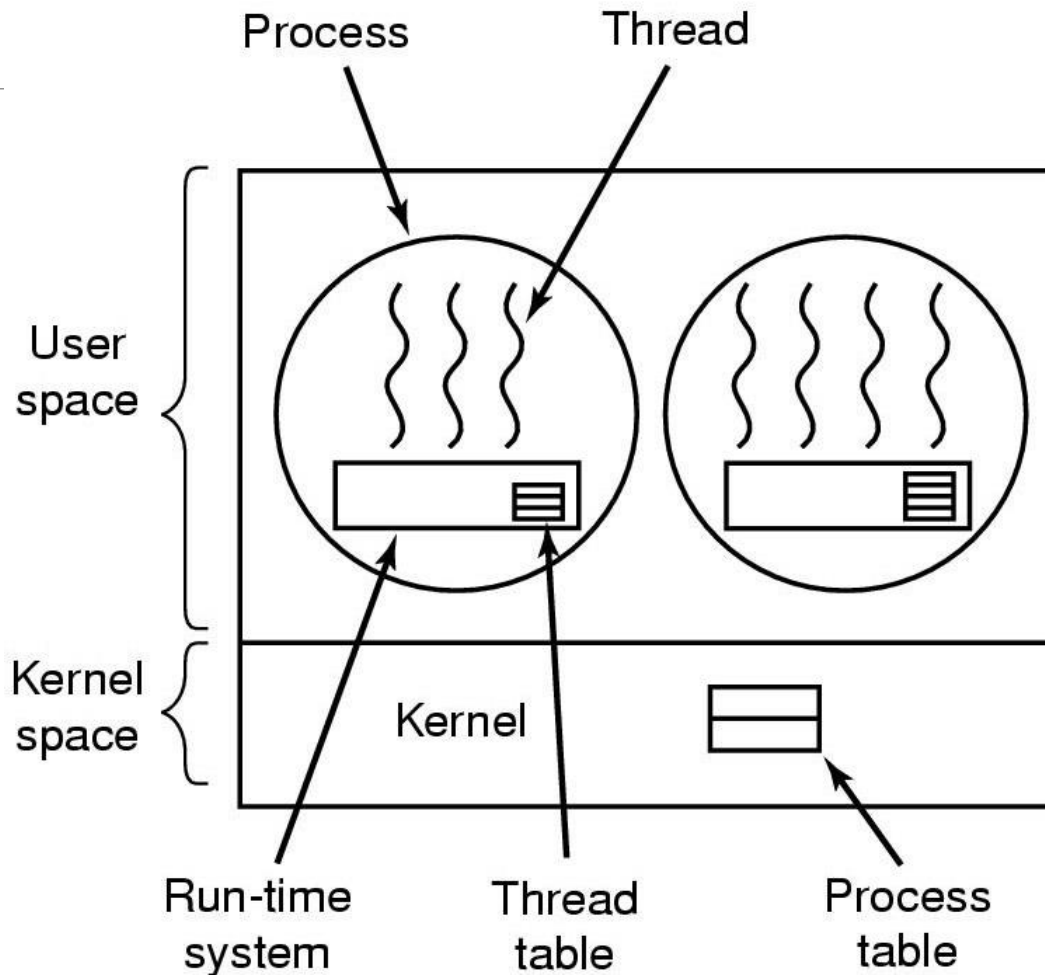
Jsou v jádře plánována vlákna nebo procesy?

Vlákná v User Space

Jádro plánuje procesy,

O vláknech nemusí vůbec vědět.

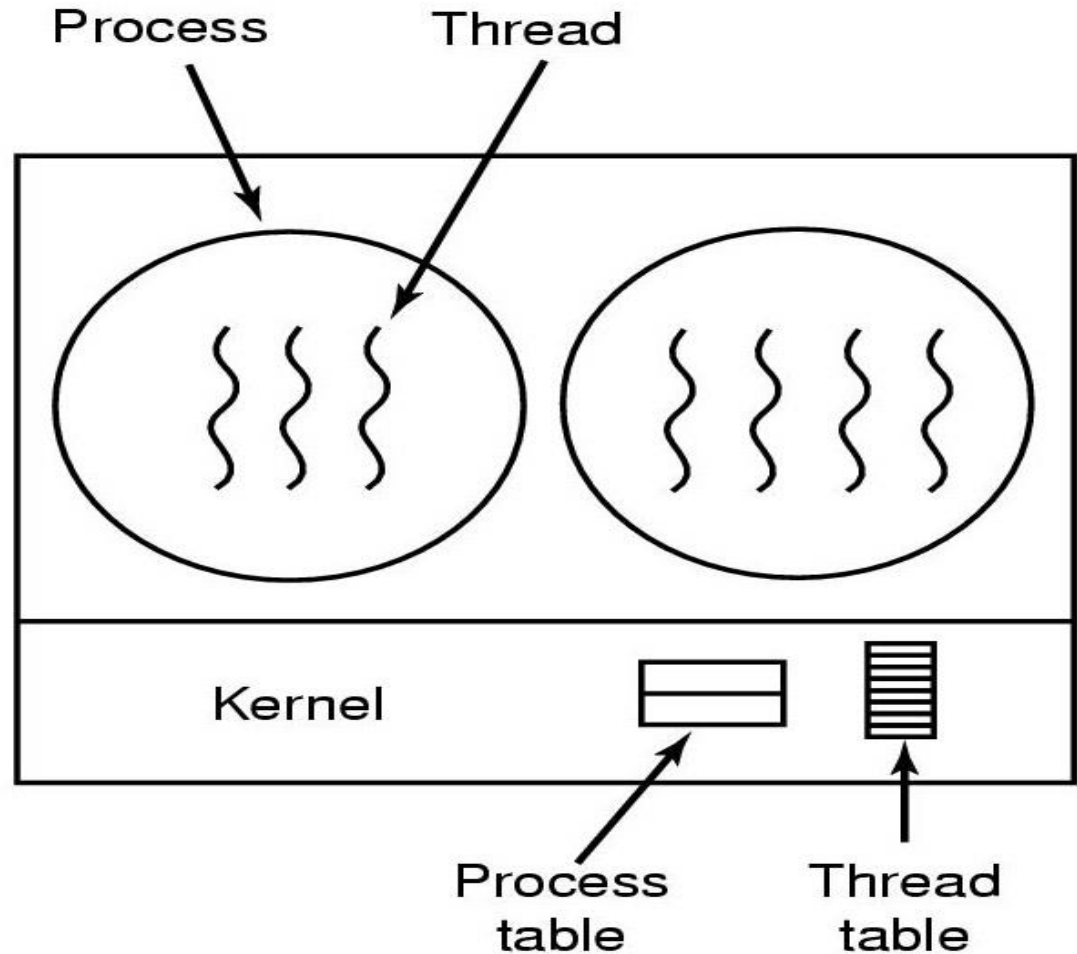
Pokud vlákno zavolá systémové volání, celý proces se zablokuje



Vlákna v jádře

Jádro
plánuje
jednotlivá
vlákna.

Kromě
tabulky
procesů má
i tabulku
vláken.

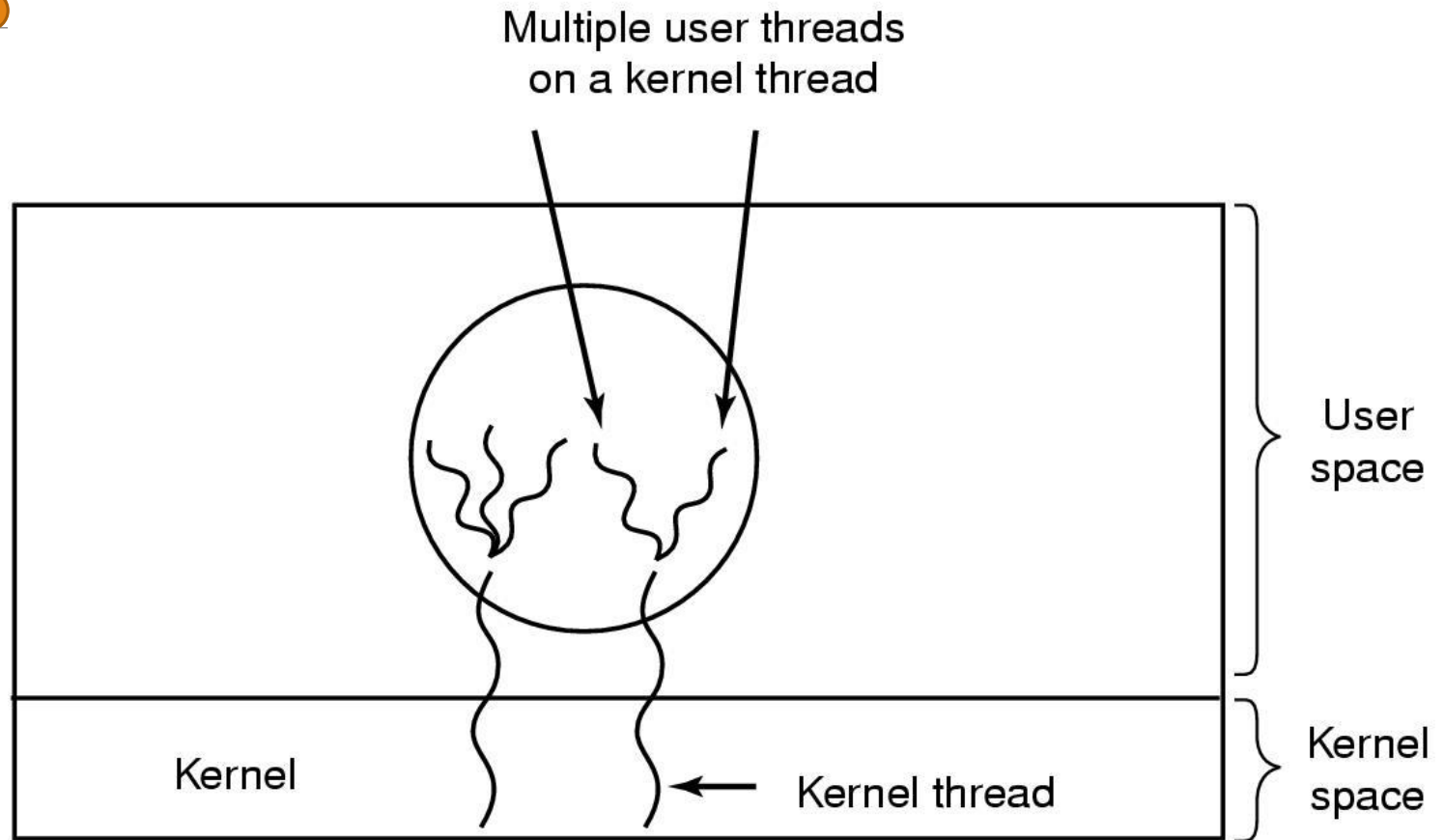


Hybridní implementace

Mapování
vláken v user
space na
vlákna v
jádře

Různá
mapování

Zobecněný
model



Modely - vlákna

one-to-one (1:1) .. vlákna v jádře

- Každé vlákno – separátní “proces” v jádře
- Plánovač jádra je plánuje jako běžné procesy
- Základní jednotkou plánování jsou vlákna

many-to-one (M:1) .. vlákna jen v user space

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

many-to-many (M:N)

- Komerční unixy (Solaris, Digital Unix, IRIX)

Vlákná - 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**

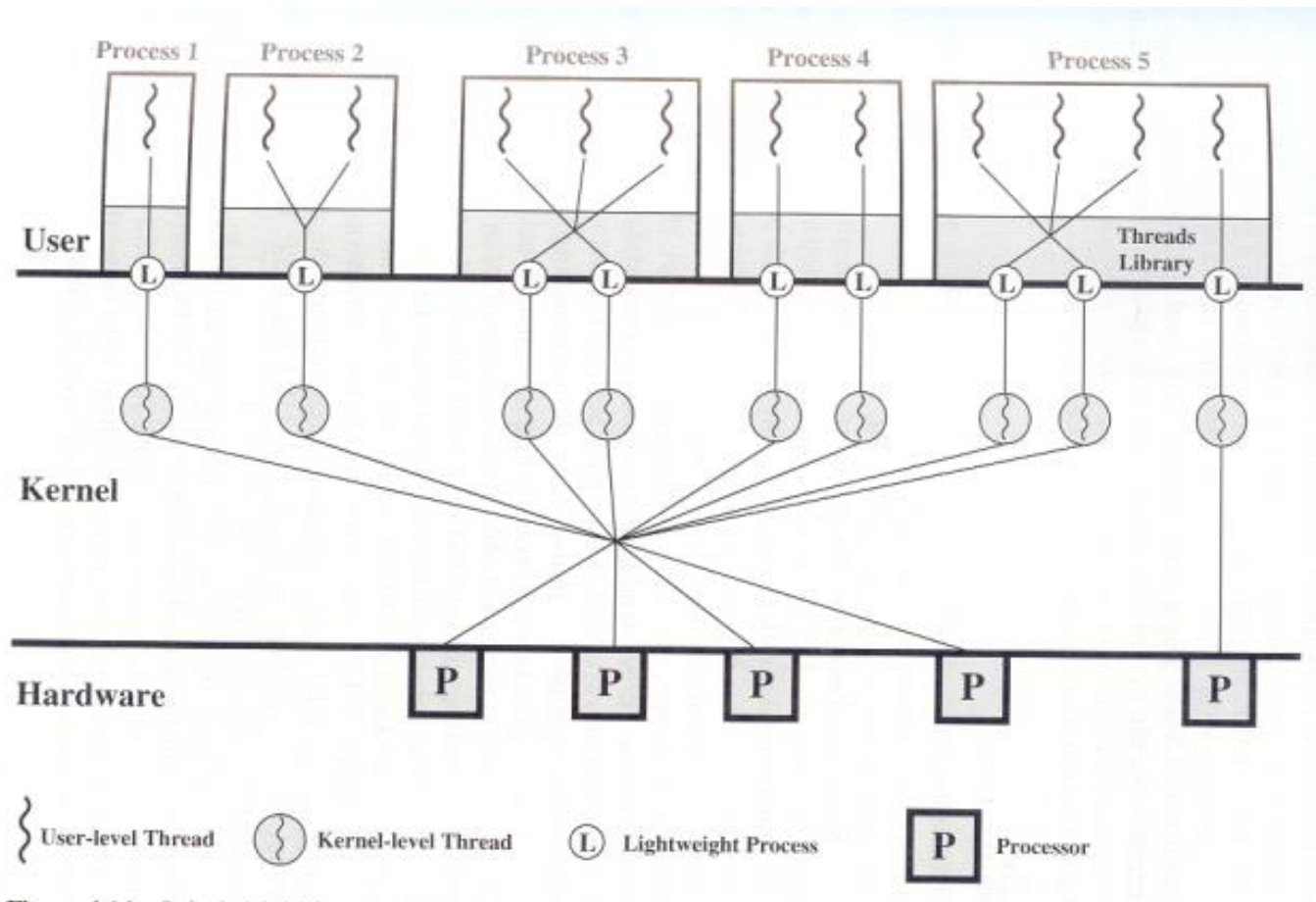
Další vlákna jádra bez LWP – např. obsluha disku

Vlákná jádra – multiplexována mezi procesory

Často uváděný příklad
obecné koncepce
vláken

Solaris – OS založený
na Unixu

Vlákna Solaris



Linux

Systémové volání **clone()**

- Zobecněná verze **forku**
- **One-to-one model**
- Dovoluje novému procesu sdílet s rodičem
 - Paměťový prostor
 - File descriptors
 - Signal handlers
- Specifické pro Linux, není přenositelné (portable), není obecně v unixových systémech

Můžeme říci, co z uvedeného bude sdíleno

Pthreads

Knihovna vláken

Historicky každý výrobce měl svoje řešení

UNIX – IEEE POSIX 1003.1c standard (1995)

- POSIX .. jednotné rozhraní, přenositelnost programů
- Implementace POSIX threads – Pthreads

`gcc -lpthread -o vlakna vlakna.c` (překlad na eryxu)

<http://yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html>

<http://www.root.cz/clanky/programovani-pod-linuxem-tema-vlakna/>

- Série článků, procesy, vlákna, synchronizace, ...

PTHREADS

Rozhraní specifikované IEEE POSIX 1003.1c (1995)

Implementace splňující tento standard:
POSIX threads , pthreads

man pthread_create

Popis v [pthread.h](#)

1. **Management vláken** ([create](#), [detach](#), [join](#))
2. **Mutexy** ([create](#), [destroy](#), [lock](#), [unlock](#))
3. **Podmínkové proměnné** ([create](#), [destroy](#), [wait](#), [signal](#))
4. **Další synchronizace** ([read-write locks](#), [bariéry](#))

Vlákná - základní funkce (!!)

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

Vlákna - synchronizační funkce

Mutex (zámek)

součástí knihovny vláken

`pthread_mutex_lock()`

- vstup do kritické sekce – „zamčení sekce“

`pthread_mutex_unlock()`

- výstup z kritické sekce – „odemčení sekce“

Implementace vláken v Linuxu

Native Posix Thread Library (NPTL)

- Využívá systémové volání `clone()`
- Synchronizační primitivum `futex`
- Implementace 1:1
 - Vlákno vytvořené uživatelem `pthread_create()` odpovídá 1:1 plánovatelné entitě v jádře (task)

Vlákná: základní funkce

#include <pthread.h>

.. knihovna vláken

pthread_t a, b;

.. id vláken a,b

pthread_create(&a, NULL, pocitej, NULL)

- **a** – id vytvořeného vlákna
- **NULL** – atributy vlákna (man pthread_attr_init)
- **pocitej** – funkce, kterou bude vlákno vykonávat
- **NULL** – argument předaný funkci pocitej
- Návratová hodnota **0** – vlákno se podařilo vytvořit

pthread_join(a, NULL);

- Čeká na dokončení vlákna s identifikátorem **a**
- Vlákno musí být v joinable state (výchozí stav, ne detach)
- **NULL** – místo null lze číst návratovou hodnotu vlákna

Příklad – vlákna – fce vlákna

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

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

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



funkce vlákna

```
void *print_message_function( void *ptr )
```

```
{
```

```
    char *message;
```

```
    message = (char *) ptr;
```

```
    printf("%s \n", message);
```

```
}
```

Příklad – vlákna - main

```
main()
```

```
{
```

```
    pthread_t thread1, thread2;
```

```
    char *message1 = "Thread 1";
```

```
    char *message2 = "Thread 2";
```

```
    int iret1, iret2;
```

```
    /* vytvoříme 2 vlákna, každé pustí podprogram s různým parametrem */
```

```
    iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
```

```
    iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);
```


Příklad – vlákna - main

```
/* hlavni vlákno bude čekat na dokončení spuštěných vláken */
```

```
/* jinak by mohlo hrozit, že skončí dřív než spuštěná vlákna */
```

```
===== zde 1 hlavní + 2 vytvořená =3 vlákna =====
```

```
pthread_join( thread1, NULL);
```

```
pthread_join( thread2, NULL);
```

```
===== zde už jen 1 hlavní vlákno =====
```

```
printf("Thread 1 returns: %d\n",iret1);
```

```
printf("Thread 2 returns: %d\n",iret2);
```

```
exit(0);
```

```
}
```

Jiný příklad: předání parametru vláknu

//vytvareni vlaken

```
for (i = 0; i < THREAD_COUNT; i++) {  
    thID = malloc(sizeof(int));  
    *thID = i + 1;  
    pthread_create(&threads[i], NULL, thread, thID);  
}
```

// funkce vlakna

```
void *thread(void * args) {  
    printf("Jsem vlakno %d\n", *((int *) args) );  
}
```

Proces UNIXU

– PCB obsahuje informace:

Proces ID, proces group ID, user ID, group ID

Prostředí

Pracovní adresář

Instrukce programu

Registry

Zásobník (stack)

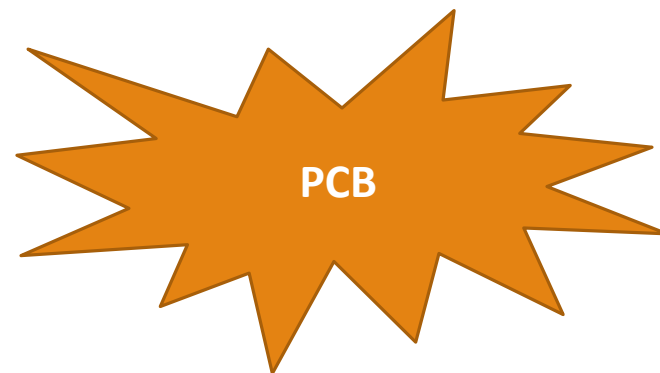
Halda (heap)

Popisovače souborů (file descriptors)

Signal actions

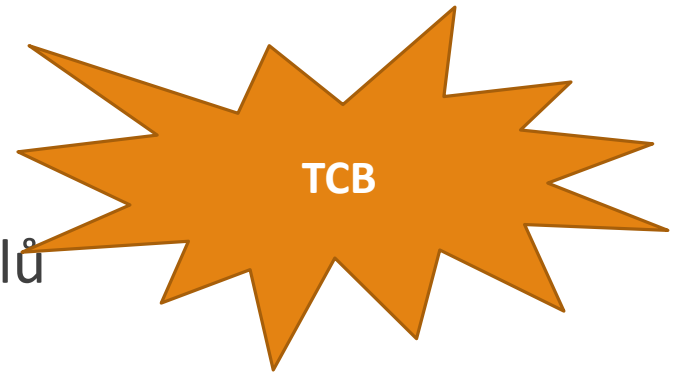
Shared libraries

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
(vlastní proměnné vlákna)



Vlákna stejného procesu sdílejí

- ❑ Adresní prostor

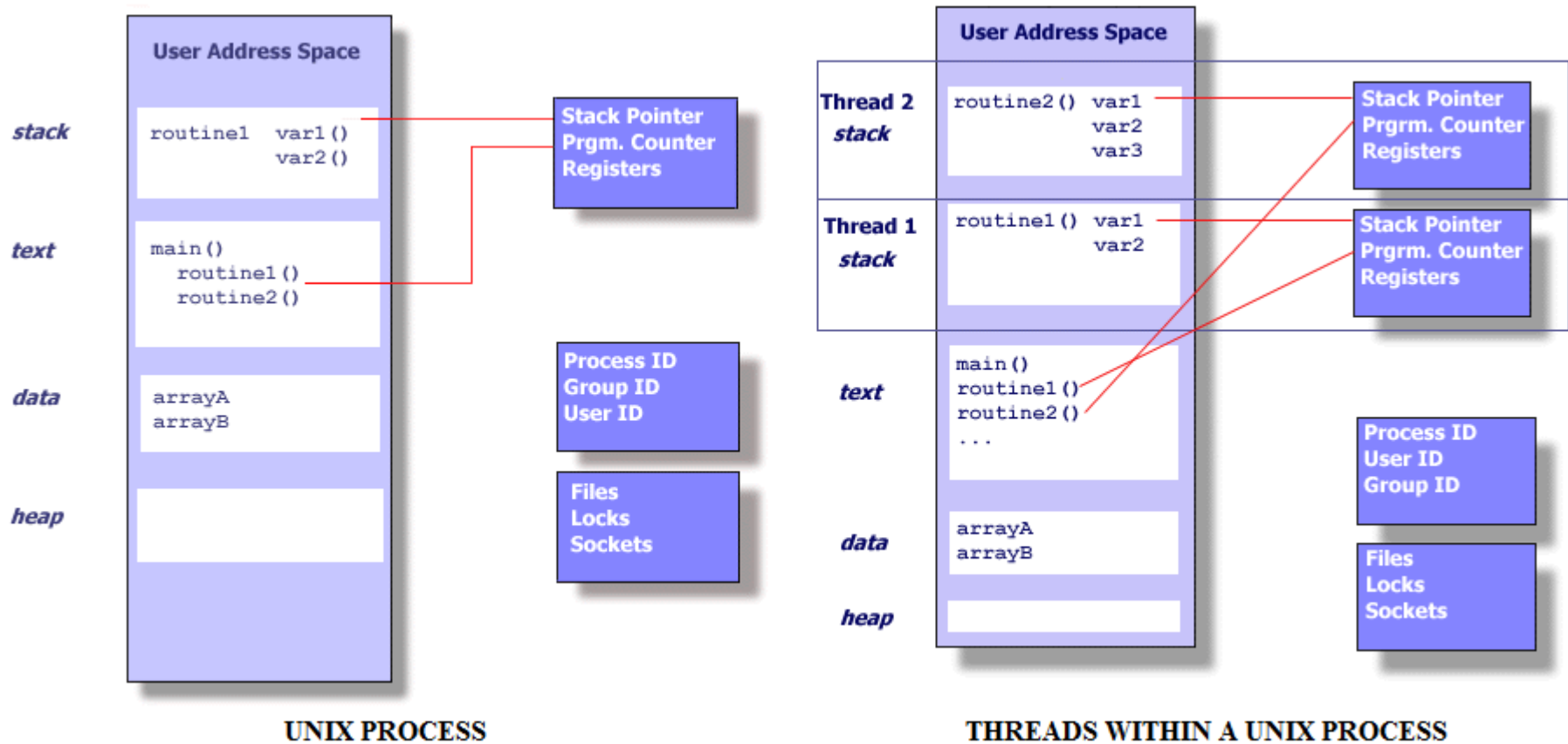
(např. globální proměnné v C sdílené napříč vlákny stejného procesu)

- ❑ Otevřené soubory

Všechna vlákna uvnitř stejného procesu sdílejí stejný adresní prostor

Mezivláknová komunikace je efektivnější a snadnější
než meziprocesová

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



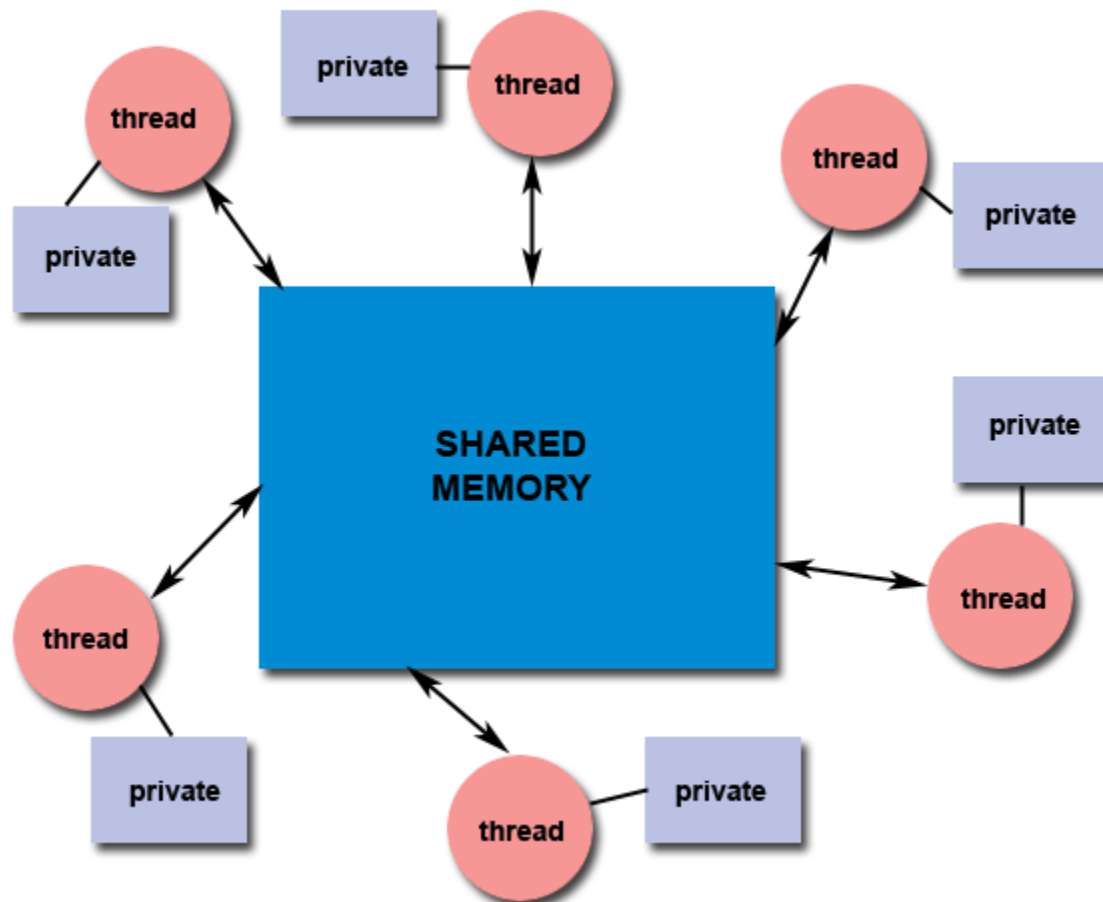
Zásobník pro vlákno

Při vytvoření vlákna můžeme specifikovat velikost zásobníku

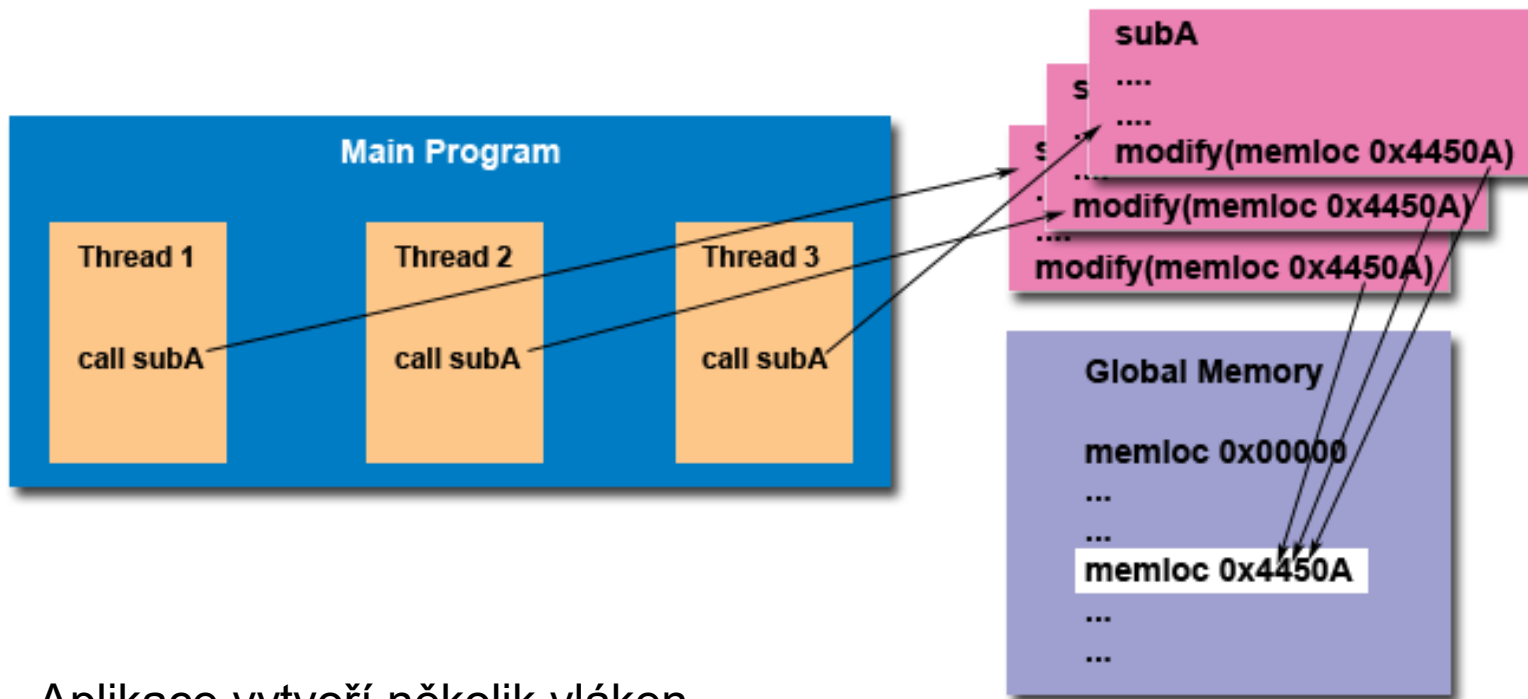
- ❑ Je potřeba celkem šetřit.
- ❑ Při max. velikost zásobníku 8MB
 - ❑ $8\text{MB} * 512 \text{ vláken} = 4 \text{ GB}$
 - ❑ Velkou část paměti by spotřeboval jen zásobník.

Globální a privátní paměť vláken

více vláken
stejného procesu



Vláknová bezpečnost (thread-safe)



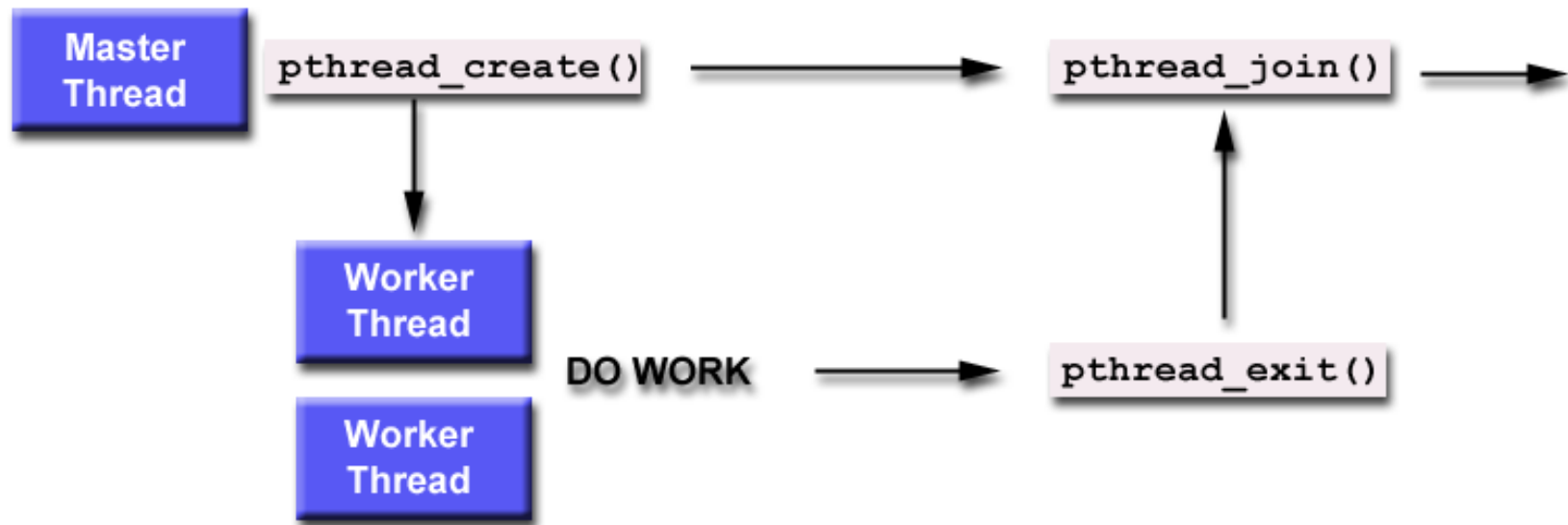
Aplikace vytvoří několik vláken

Každé vlákno vyvolá stejnou rutinu

Tato rutina modifikuje společná globální data

– pokud nemá synchronizační mechanismy, není thread-safe

Čekání na dokončení vláken



Možnosti ukočení vlákna

Vlákno dokončí „proceduru vlákna“ - nejčastější

Vlákno kdykoliv zavolá `pthread_exit()`

Vlákno je zrušené jiným vláknem `pthread_cancel()`

PROCES zavolá `execve()` nebo `exit()`

Pokud `main()` skončí první bez explicitního volání `pthread_exit()`

Vlákna - Java

Vlákno – instance třídy `java.lang.Thread`

Odvodit potomka, překrýt metodu `run()`

- Vlastní kód vlákna

Spuštění vlákna – volání metody `start()` třídy `Thread`

Další možnost - třída implementující rozhraní `Runnable`

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

Problémy preemptivních systémů

Pokud je systém **preemptivní** (což často chceme, aby se procesy rychle střídaly na CPU), může dojít k odstavení procesu od procesoru v **nevhodný čas**

Např. manipuluje se **sdílenou** datovou strukturou, a než dokončí všechny potřebné akce, dojde k přeplánování na jiný proces (vlákno), což může vést ke špatnému výsledku

Taková chyba se může projevit velmi **nepravidelně**, třeba 1x za 100 000 běhů programu.

Synchronizace procesů

Časový souběh

Kritická sekce

Algoritmy pro přístup do kritické sekce

Semaforey

Důležité pojmy, které je potřeba znát a umět vysvětlit

Časový souběh (!!)

Procesy mohou sdílet **společnou paměť** pro čtení a zápis.
Vlákna jednoho procesu také sdílejí paměť.

Může nastat **časový souběh** (**race condition**):

Časový souběh = dva nebo více procesů či vláken se pokusí současně přistoupit ke stejným zdrojům a výsledkem může být chyba (např. špatná hodnota sdílené proměnné)

Zdrojem rozumíme např. sdílené proměnné.

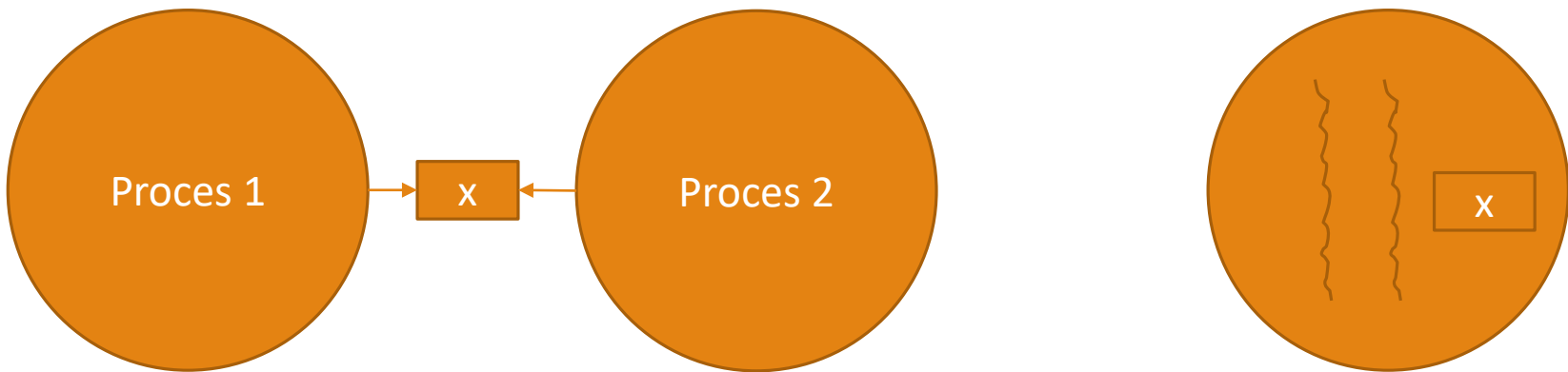
Klasický příklad:

dva procesy zvětšují asynchronně sdílenou proměnnou X

Jak na sdílenou paměť?

Procesy požádají operační systém o přidělení úseku sdílené paměti - voláním `shmget()`

Vlákna stejného procesu paměť sdílejí



Příklad dvou procesů

cobegin

...

$x = x + 1;$

...

||

...

$x = x + 1;$

...

coend



1.proces



2.proces

společná
paměť:

int x;

1 příkaz - převod na instrukce CPU (!)



$x = x + 1;$

1. Načtení hodnoty x do registru (LD R, x)
2. Zvýšení hodnoty x (INC R)
3. Zápis nové hodnoty do paměti (LD x, R)

Pokud oba procesy provedou všechny tři příkazy **sekvenčně**,
bud mít x správně **$x+2$**

Chybné pořadí vykonání

Přepnutí plánovače v nevhodném okamžiku. (pseudoparalelní běh)

1. P1: LD R, x // x je 0, R je 0
2.  P2: LD R, x // x je 0, R je 0
3. INC R // x je 0, R je 1
4.  LD x, R // x je 1, R je 1
5. P1: // x je 1, R je 0 – rozpor
6. INC R // x je 1, R je 1
7. LD x, R // x je 1, R je 1

Výsledek – **chyba**, neprovedlo se obojí zvětšení, v x je 1 místo správné hodnoty 2.

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
```

K chybě může dojít jak při pseudoparalelním běhu,
tak i při paralelním běhu

Př. bankovní transakce

Dva procesy přístup do databáze

Účet := účet + 20 000 1. proces

Účet := účet – 15 000 2. proces

Správný výsledek?

Možné výsledky?

Časový souběh – další příklady

Přidávání prvku do seznamu

- Častá činnost v systémovém programování

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** (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 (critical section)

Kritická sekce je místo v programu, kde je prováděn přístup ke společným datům.

Procesy, vlákna – komunikují přes společnou datovou oblast
(sdílená paměť – procesy, globální proměnné – vlákna)

Cílem je zařídit, aby byl v kritické sekci v daný okamžik pouze JEDEN proces (vlákno).

Společná datová oblast

hlavní paměť (sdílené proměnné x, y, z, \dots)

Soubor s_1

- Souběžně do něj zapisuje a čte více procesů.
- **zamykání částí souboru** – řeší časový souběh

každá **kritická sekce** se vztahuje ke **konkrétním datům**, ke kterým se v ní přistupuje (x, y, z, s_1, \dots)

Počet kritických sekcí

Kritická sekce nemusí být jedna

Pokud procesy sdílejí tři proměnné x , y , z

- Každá z nich představuje KS_x , KS_y , KS_z

Mohli bychom sice říci, že jde o jednu KS, ale potom bychom zbytečně blokovali přístup k y , řešíme-li souběh nad x atd.

Analogie: když potřebujeme zamknout řádku tabulky v databázi, není potřeba zamykat celou tabulku, která může mít třeba milion záznamů – vliv na výkon systému

Struktura procesů

cobegin

P1: while true do

// nekonečná smyčka

begin

nevinná_činnost;

// pouze s **vlastními** daty

kritická_sekce

// přístup do **sdílených** dat

end

||

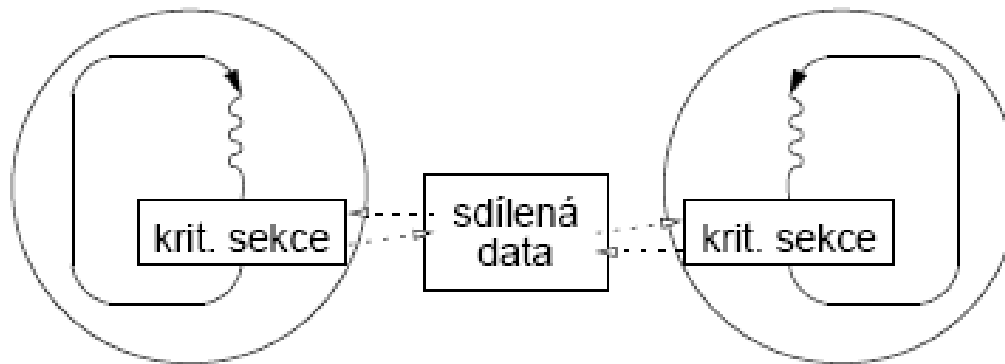
P2: ...

// totéž co P1

coend

Cílem slidu je říci, že činnost procesu se skládá z částí, kdy pracuje s vlastními daty a z částí, kdy přistupuje ke sdíleným datům

Kritická sekce



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

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 (!!!!)

1. Zákaz přerušení
2. Aktivní čekání
3. Zablokování procesu

Zákaz přerušení

vadí nám přeplánování procesů

- může nastat v nevhodný čas
- k přeplánování dojde díky **přerušení (od časovače)**

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**
(uživatel by zakázal přerušení a už třeba nepovolil...)

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
- Tedy 2 procesy chtějí přistoupit -> výsledek sekvenční provedení CPU instrukcí
- 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;           // pseudokod
int turn;
begin
  turn = 1;
  cobegin
    P1: while true
      {
        while turn == 2 ;           // čekací smyčka
          KS;                         // kritická sekce
        turn = 2                     // a může druhý
      }
  endcobegin
end
```

Algoritmus 1 pokračování

||

P2: while true

{

while **turn == 1** ; // čekací smyčka

KS; // **kritická** sekce

turn == 1 // a může první

}

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
- Poté čekání skončí

Nevýhoda aktivního čekání

- **plýtvá** časem CPU (neustálé testování hodnoty)

Kdy lze použít

- **když předpokládáme krátké čekání**
- **spin lock** (“vrtit se nad zámkem”)

Algoritmus - Peterson

První úplné řešení navrhl Dekker, ale je poměrně složité

Jednodušší a elegantnější algoritmus navrhl Peterson (1981)

- Uvedeno řešení pro 2 procesy
- lze ale zobecnit pro více procesů

Peterson – enter_CS()

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

Peterson – leave_CS()

```
void leave_CS(int process) {  
  
    interested[process] = false;    // oznámí odchod z KS  
  
}
```

Peterson – použití enter_CS() a leave_CS()

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.

Z funkce **enter_CS** se vrátí až tehdy, když je kritická sekce volná !

Zavoláním **leave_CS** dáme najevo, že naše kritická sekce končí a dovnitř může někdo další

Peterson - vysvětlení

while (**turn==process**) and (**interested[other]==true**) do ;

Pokud chce do KS **pouze jeden** z procesů:

interested[other] bude false, a smyčka končí

Pokud chtějí do KS **oba dva**:

rozhoduje první část `turn == process`

turn bude vždy mít hodnotu 0, nebo 1, nic jiného
jeden z procesů skončí čekací smyčku a může do KS

Peterson – vysvětlení podrobněji

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**, výsledek bude buď 0 nebo 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:

CPU instrukce, která **otestuje hodnotu a nastaví paměťové místo v jedné nedělitelné operaci**

operace **Test and Set Lock** – TSL:

TSL R, lock

- LD R, lock // do R načte hodnotu buňky lock
- LD lock, 1 // do paměť. buňky lock uloží 1

(R je registr CPU)

(lock – buňka paměti, obsahuje buď 0 false nebo 1 true)

TSL

Provádí se **nedělitelně (atomicky)** – žádný proces nemůže k proměnné lock přistoupit do skončení TSL

Multiprocesor – **zamkne** paměťovou **sběrnici** po dobu provádění instrukce

TSL - použití

Proměnná typu **zámek (lock)** – na počátku 0 = odemčeno

Proces, který chce vstoupit do KS – test

- Pokud 0, nastaví na 1 (zamčeno) 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 provede R<-lock, lock<-1
CMP R, 0	:: byla v lock 0, tedy odemčeno?
JNE spin_lock	:: pokud ne cykluj dál
RET	:: návrat, vstup do KS

Spin_unlock:

LD lock, 0	:: ulož hodnotu 0 (odemčeno) do lock
RET	

Implementace zámku – pozn.

Cyklus přes návěští `spin_lock` dokud lock je **1** (zamčeno)

Když někdo jiný 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

Poznámka

Proč se používá tento delší kód?

TSL je drahá operace (ve smyslu náročná na zdroje)

CMP je méně náročná operace

Lepší je častěji zkoušet méně náročnou operaci

Náhrada TSL – když by nebyla

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 – v pseudokódu

```
boolean atomic function TSL (boolean zamceno) {  
    int pom;  
  
    pom = zamceno;  
    zamceno = true;  
    return pom;  
}
```

Instrukci TSL si můžeme namodelovat s využitím atomické funkce (provede se nedělitelně)

Implementace spin-locku

```
boolean lock;
```

```
void spin_lock (var m: lock) {
```

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

```
}
```

Implementace spin-locku

```
void spin_unlock (var m: lock);  
{  
    m = false;  
}
```

Pozn. V literatuře TSL někdy se nastavuje true, někdy false; chce to předem znát sémantiku, co se označuje jako odemčeno a co jako zamčeno.

Problém řešení s aktivním čekáním

(Peterson, spin-lock)

Ztracený čas CPU

- Jeden proces v KS, další může ve smyčce přistupovat ke společným proměnným
 - **krađe paměťové cykly aktivnímu procesu**

Problém inverze priorit

- Pokud procesy mají prioritu
- Dva procesy, jeden s **vyšokou H** a druhý s **nížskou L** prioritou, H se spustí jakmile připraven

Problém inverze priorit

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**

Řešení problémů s akt. čekáním

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

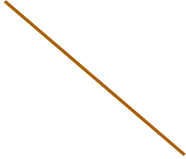
Nad semaforey **pouze operace** **P(s)** a **V(s)**

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 zablokováný, 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

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 dokončení KS – $V(s)$

$P(s); \dots \text{KS} \dots ; V(s);$

Je-li libovolný proces v KS

- Potom S je 0, jinak S je 1

Vzájemné
vyloučení

Do kritické sekce
smí vstoupit
pouze 1 proces
současně

Na počátku je
vstup do kritické
sekce volný

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

Otázky k uvedenému příkladu

Co kdybychom na začátku semafor špatně inicializovali na hodnotu 2?

Co kdybychom na začátku semafor špatně inicializovali na hodnotu 0?

Co kdybychom zapomněli po dokončení kritické sekce zavolat $V(s)$?

Co kdybychom před vykonáním kritické sekce nezavolali operaci $P(s)$?

Použití semaforů

Binární semafor může nabývat jen hodnot 0 a 1

Vzájemné vyloučení

- **Mutexy**, binární semafore .. 0 a 1

Kooperace procesů

- Problém omezených zdrojů (např. velikost bufferu)
- Obecné semafore .. 0, 1, 2 ..

Pro vzájemné vyloučení můžeme samozřejmě využít obecný semafor, binární nám navíc může ohlídat, že hodnoty budou jen 0 a 1

Vzájemné vyloučení - KS

Procesy **soupeří** o zdroj

Ke zdroji může chtít 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

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.

Problém producent-konzument

Problém ohraničené vyrovnávací paměti
(bounded buffer problem, Dijkstra 1968)



Producent
a
Konzument

běží paralelně

Dva procesy sdílejí
společnou paměť (buffer) **pevné velikosti N** položek

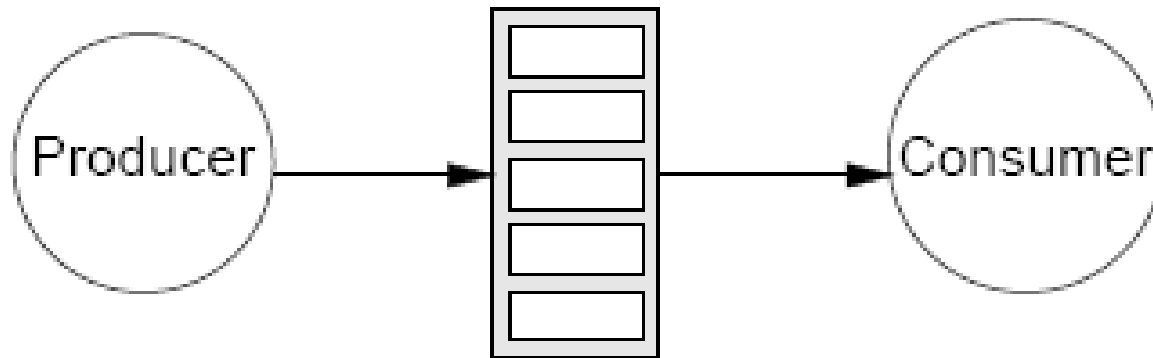
Proces **producent**

- **generuje** nové položky a ukládá je do společné paměti

Proces **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
(zapsání do plného bufferu, čtení z prázdného bufferu)

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 konzumenty (full)

m - pro **vzájemné vyloučení** – práce se společnou pamětí – **kritická sekce**

P&K - implementace

semaphore

```
e = N;           // počet prázdných položek  
f = 0;           // počet plných položek  
m = 1;           // vzájemné vyloučení (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 počet 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;

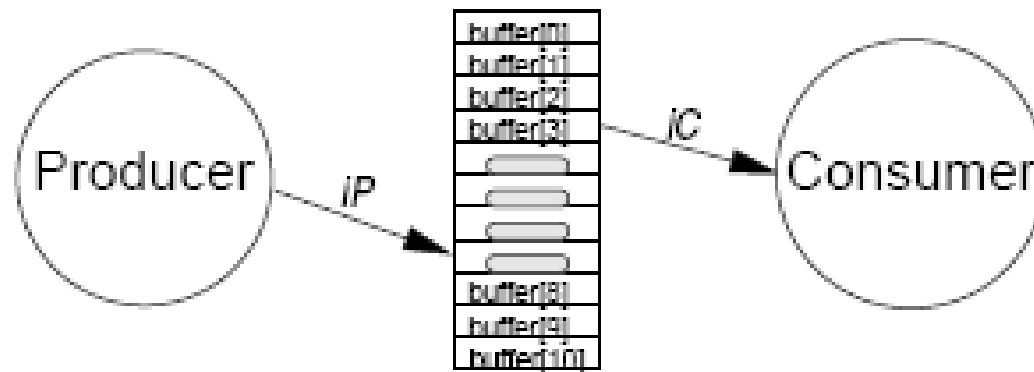
end {while}

coend.

Pokud je buffer prázdný,
zablokuje se

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) mod 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