EVROPSKÁ UNIE
Evropské strukturální a investiční fondy
Operační program Výzkum, vývoj a vzdělávání

MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

Operační systémy

Komunikace mezi procesy

IPC- Inter Process Communication

Strategický projekt UTB ve Zlíně, reg. č. CZ.02.2.69/0.0/0.0/16_015/0002204

Martin Sysel
Fakulta aplikované informatiky
Univerzita Tomáše Bati ve Zlíně

Spolupráce mezi procesy

- Nezávislý proces - pokud nemůže ovlivnit nebo být ovlivněn jinými procesy
  - Každý proces, který nesdílí data s žádným jiným procesem, je nezávislým procesem.
- Spolupracující proces - pokud může ovlivnit nebo být ovlivněn jinými procesy
  - Jakýkoli proces, který sdílí data s jinými procesy, je spolupracujícím procesem.

  - Výhody spolupracujících procesů
    - Sdílení informací
    - Zrychlení výpočtu
      - Pokud má počítač více jader, je možné rozdělit úkol na dílčí úkoly a provádět je paralelně.
    - Modularita
      - Modulární systém umožňuje rozdělit systémové funkce do samostatných procesů nebo vláken.
    - Pohodlí
      - Dokonce i jeden uživatel může pracovat na mnoha úkolech současně.

(Silberschatz, Galvin & Gagne, 2013)

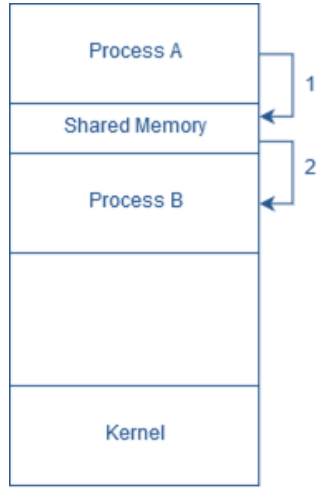
Meziprocesová komunikace

- **InterProcess Communication (IPC)**
- Mechanismus, který umožňuje vzájemnou komunikaci a synchronizaci akcí.
  - Komunikace mezi procesy je metoda spolupráce mezi nimi.
- Procesy mohou spolu komunikovat různými způsoby
  - Sdílená paměť (Shared Memory)
  - Předávání zpráv (Message Passing)
    - Roury (Pipes)
    - Sokety (síťové nebo uvnitř jádra)
  - Soubory
- Synchronizace procesů (vláken) nebo spouštění akcí jinými procesy.
  - Signály
  - Semafory
  - Spinlocks
  - …
- Vzdálené volání procedur (RPC)

(Silberschatz, Galvin & Gagne, 2013; Stallings, 2014)

Sdílená paměť

- Nejrychlejší forma meziprocesové komunikace
- Jedná se o společný blok virtuální paměti sdílené více procesy.
- Procesy čtou a zapisují do sdílené paměti pomocí stejných instrukcí, jaké používají pro přístup do vlastního prostoru virtuální paměti.
  - Oprávnění je pro proces jen pro čtení nebo pro čtení a zápis
  - Komunikace je pod kontrolou uživatelských procesů, nikoli OS.
    - Zabezpečení přístupu (vzájemné vyloučení - Mutual exclusion) nejsou součástí správy sdílené paměti, ale musí být poskytováno samotnými procesy.



(Stallings, 2014)

Zápis do sdílené paměti

```
#include <iostream>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
using namespace std;

int main()
{
    // ftok to generate unique key
    key_t key = ftok("shmfile",65);

    // shmget returns an identifier in shmid
    int shmid = shmget(key,1024,0666|IPC_CREAT);

    // shmat to attach to shared memory
    char *str = (char*) shmat(shmid,(void*)0,0);

    cout<<"Write Data : ";
    gets(str);

    printf("Data written in memory: %s\n",str);

    //detach from shared memory
    shmdt(str);

    return 0;
}
```

Čtení ze sdílené paměti

```
#include <iostream>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
using namespace std;

int main()
{
    // ftok to generate unique key
    key_t key = ftok("shmfile",65);

    // shmget returns an identifier in shmid
    int shmid = shmget(key,1024,0666|IPC_CREAT);

    // shmat to attach to shared memory
    char *str = (char*) shmat(shmid,(void*)0,0);

    printf("Data read from memory: %s\n",str);

    //detach from shared memory
    shmdt(str);

    // destroy the shared memory
    shmctl(shmid,IPC_RMID,NULL);

    return 0;
}
```

(https://www.geeksforgeeks.org/ipc-shared-memory/)

Předávání zpráv (Message Passing)

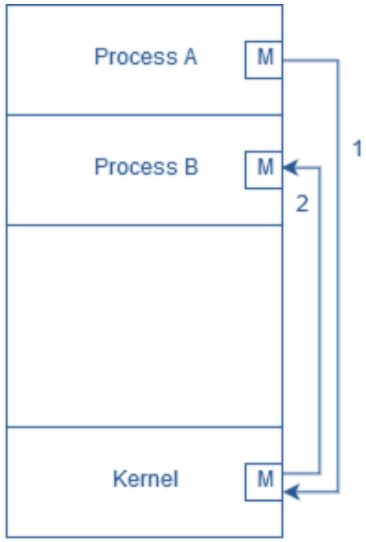
- Systémy pro předávání zpráv mohou mít mnoho podob.
  - obvykle ve formě dvojice primitivních operací (primitives):

odeslat (cíl, zpráva) *send (destination, message)*

- Proces odešle informace ve formě zprávy jinému procesu označenému jako cíl.

přijmout (zdroj, zpráva) *receive (source, message)*

- Proces přijímá informace ze zdroje



Message Passing
s použitím fronty zpráv

- Procesy vytvoří komunikační spojení a mohou komunikovat

(Silberschatz, Galvin & Gagne, 2013; Stallings, 2014)

Návrhové charakteristiky systému zpráv pro IPC

- Synchronizace
  - Odesílání a přijímání může být blokováno nebo neblokováno (synchronní, asynchronní)
- Ukládání do vyrovnávací paměti
  - Nulová kapacita, omezená nebo neomezená kapacita fronty.
- Adresování
  - Přímé - odesílání nebo příjem (automatické nebo explicitní „ruční“)
  - Nepřímé - statické nebo dynamické nebo vlastnictví. (mail-box)
- Formát
  - Obsah - pouze zpráva nebo záhlaví a tělo
  - Délka - pevná nebo variabilní
- Řazení ve frontě
  - FIFO (nestačí, pokud jsou některé zprávy naléhavější než jiné) nebo Priorita

(Silberschatz, Galvin & Gagne, 2013)

Návrhové charakteristiky systému zpráv pro IPC

- Další možné problémy s implementací:
  - Jak je vytvářeno spojení?
  - Může být spojení využito více procesy?
  - Kolik spojení může být využito párem procesů?
  - Je odkaz jednosměrný nebo obousměrný?
- Ošetření chyb při zasílání zpráv musí zahrnovat tyto situace:
  - jeden z partnerských procesů skončil
  - ztráta zprávy
  - duplicita zprávy
  - zkomolení zprávy

Synchronizace systému zpráv

- Předávání zpráv může být blokující nebo neblokující
- Blokující je synchronní
  - Blokující odeslání: odesílatel je zablokován, dokud zprávu nepřijme druhá strana
  - Blokující příjem: příjemce je blokován, dokud není k dispozici zpráva
  - Rendezvous: blokující dokud příjemce nepřijme a nepotvrdí zprávu
- Neblokující je asynchronní
  - Neblokující odesílání: odesílatel odešle zprávu a pokračuje ve provádění
  - Neblokující příjem: přijemce obdrží platnou zprávu nebo nulovou zprávu (pokud do příjemce nebylo nic zasláno)
- Častou kombinací je neblokující odesílání a blokující příjem.

(Deitel, Deitel & Choffness, 2004; Silberschatz, Galvin & Gagne, 2013)

Ukládání do vyrovnávací paměti

- Žádné ukládání do vyrovnávací paměti
  - Odesílatel musí počkat, než příjemce přijme zprávu
  - Rendezvous u každé zprávy
- Omezená velikost vyrovnávací paměti
  - Konečná velikost
  - Odesílatel je blokován pokud je vyrovnávací paměť plná
  - K vyřešení problému lze použít monitor
- Neomezená velikost
  - Odesílatele není třeba nikdy blokovat

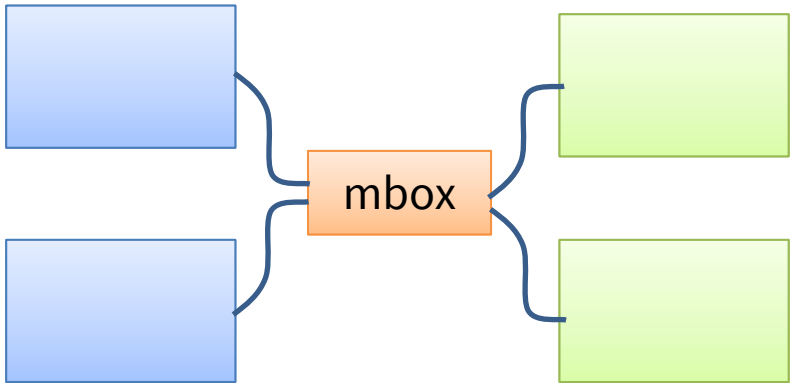
(Deitel, Deitel & Choffness, 2004; Silberschatz, Galvin & Gagne, 2013)

Přímé spojení

- Jedna vyrovnávací paměť v přijímači
  - Více než jeden proces může odesílat zprávy příjemci
  - Pro příjem od konkrétního odesílatele je nutné prohledat celou vyrovnávací paměť
- Vyrovnávací paměť u každého odesílatele
  - Odesílatel může odesílat zprávy více příjemcům
  - Pro získání zprávy je také nutné prohledat celou vyrovnávací paměť

(Deitel, Deitel & Choffness, 2004; Silberschatz, Galvin & Gagne, 2013)

Nepřímé spojení



- Jako abstrakce se použije poštovní schránka
  - Umožňuje mnohostrannou komunikaci
  - Větší flexibilita
    - One-to-one (privátní spojení), Many-to-one (client/server), One-to-many (broadcast)
- Ukládání do vyrovnávací paměti
  - Vyrovnávací paměť a synchronizace by měly být implementovány v poštovní schránce
- Velikost zprávy
  - Velkou zprávu lze rozdělit na pakety

(Deitel, Deitel & Choffness, 2004; Silberschatz, Galvin & Gagne, 2013)

Roury (Pipes)

- Komunikace mezi dvěma souvisejícími procesy.
- Umožňuje tok dat pouze v jednom směru.
  - Mechanismus je half duplex, což znamená, že první proces komunikuje s druhým procesem.
  - Data zapsaná do potrubí jsou operačním systémem ukládána do vyrovnávací paměti, dokud nejsou načtena z konce potrubí.
- K dosažení full duplex je vyžadováno další potrubí
- Pojmenovaná roura (named pipe) je implementována prostřednictvím souboru v systému souborů
  - Jiný název pro FIFO
  - Lze použít pro obousměrnou komunikaci
  - Soubor lze použít jako vyrovnávací paměť pro dva nebo více nesouvisejících procesů

(Deitel, Deitel & Choffness, 2004; Silberschatz, Galvin & Gagne, 2013)

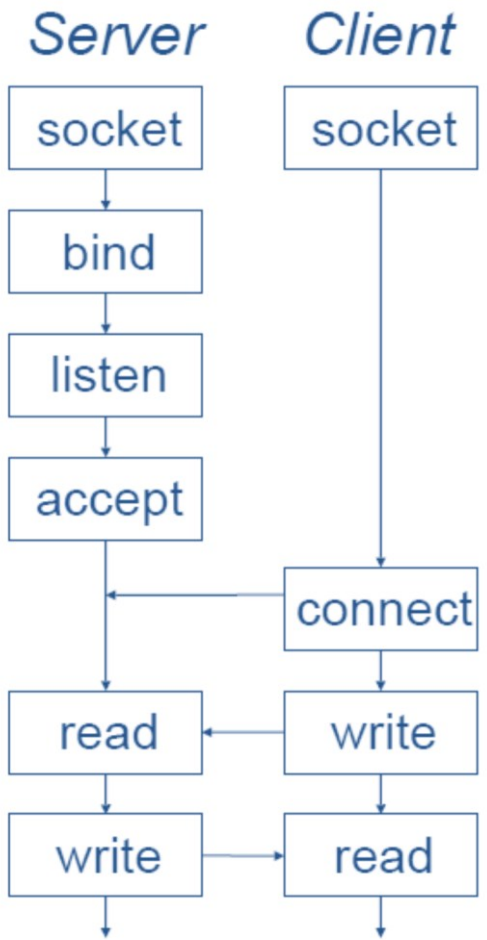
Sokety

- Soket je definován jako koncový bod pro komunikaci.
  - Dvojice komunikačních procesů používá pár soketů - jeden pro každý proces.
- Data odesílaná přes síťové rozhraní
  - buď do jiného procesu na stejném počítači nebo do jiného počítače v síti.
  - TCP, UDP, …
- Unix domain socket
  - Podobně jako síťový soket, ale veškerá komunikace probíhá v jádře.
  - Doménové sokety používají systém souborů jako svůj adresní prostor.
  - Procesy odkazují na doménové sokety na i-uzel (i-node)
    - více procesů může používat ke komunikaci jeden soket

(Deitel, Deitel & Choffness, 2004; Silberschatz, Galvin & Gagne, 2013)

Sockety

- Abstrakce pro TCP a UDP
- Adresování
  - IP adresa a číslo portu
- Vytvoření a uzavření socketu
  - socket (af, typ, protokol);
  - Sockerr = close (sockid);
- Navázání socketu na místní adresu (bind)
  - sockerr = bind (sockid, localaddr, addrlength);
- Vyjednat připojení
  - Listen (sockid, délka);
  - accept(sockid, addr, délka);
- Připojení socketu k cíli
  - connect (sockid, destaddr, addrlength);



(Singh, 2018)

Soubory

- Záznam uložený na disku nebo záznam syntetizovaný na vyžádání souborovým serverem
  - Mohou být přístupné několika procesy zároveň (čtení).

- Soubor mapovaný v paměti
  - Soubor mapovaný v RAM (segment virtuální paměti)
  - Můžeme přistupovat přímo na adresu v paměti místo streams.
    - Stejné výhody jako standardní soubor.

Signály

- Asynchronní, jednosměrná systémová zpráva odeslaná z jednoho procesu do druhého.
  - příjemcem signálu je pouze proces, odesílatel je buď proces, nebo jádro OS
- Nepoužívá se k přenosu dat, ale pro dálkové ovládání partnerského procesu.
  - zaslání jednoduché zprávy (nastavení 1 bitu), která je definována číslem signálu
- Specifikováno ve standardu POSIX.

Linuxové signály

- Linuxové jádro implementuje asi 30 signálů.
  - Každý signál je označen číslem od 1 do 31.
- Procesy mohou řídit, co se stane, když obdrží signál.
  - S výjimkou SIGKILL a SIGSTOP, který proces vždy ukončí nebo zastaví.
- Proces může
  - přijmout výchozí akci, což může být ukončení procesu, ukončit a vypsat core dump procesu, zastavit proces
  - Rozhodnout, zda bude ignorovat nebo zpracovávat signály v závislosti na typu signálu.
    - Ignorované signály jsou zahozeny.
    - Zpracovávané signály spouští uživatelskou obsluhu signálu.
      - Program skočí na tuto funkci, jakmile je signál přijat
      - Potom se dále pokračuje od přerušené instrukce
- Termín raise se používá k označení generování signálu
- Termín catch se používá k označení přijetí signálu.

(Linux Manual pages)

Linuxové signály

- Signál (v základní verzi) je číslo (int) zaslané procesu prostřednictvím rozhraní (pro tento účel definovaného).
- Signály jsou generovány
  - při chybách (např. aritmetická chyba, chyba práce s pamětí, ...),
  - externích událostech (vypršení časovače, dostupnost I/O, ...),
  - na žádost procesu – IPC (kill, ...).
- Signály často vznikají asynchronně k činnosti programu
  - není možné jednoznačně předpovědět, kdy signál bude doručen.

(Vojnar, 2011)

Linuxové signály

- SIGHUP– odpojení, ukončení terminálu
- SIGSTOP, SIGTSTP– tvrdé/měkké pozastavení (Ctrl-Z)
- SIGCONT– pokračuj, jsi-li pozastaven
- SIGINT– přerušení z klávesnice (Ctrl-C)
- SIGTERM– měkké ukončení
- SIGKILL– násilné (tvrdé) ukončení
- SIGSEGV, SIGBUS– chybný odkaz do paměti
- SIGPIPE– zápis do roury bez čtenáře
- SIGALRM– signál od časovače (alarm)
- SIGUSR1, SIGUSR2– uživatelské signály
- SIGCHLD– pozastaven nebo ukončen potomek

(man 7 signal; Vojnar, 2011)

Problémy souběžného vykonávání procesů

- Souběžný přístup ke sdíleným datům může vést k nekonzistenci dat
  - Udržování konzistence dat vyžaduje koordinaci, která zajistí řádný běh spolupracujících procesů
- Synchronizace běhu procesů
  - Čekání na událost způsobenou jiným procesem.
- Mezi procesová komunikace (IPC)
  - Výměna informací (zpráv)
  - Synchronizace, koordinace různých činností
- Sdílení zdrojů
  - Souběh (Race condition)
  - Uváznutí (Deadlock)

(Silberschatz, Galvin & Gagne, 2013)

Souběh (Race condition)

- Situace, kdy několik procesů přistupuje ke sdíleným datům současně a manipuluje s nimi.
  - Konečná hodnota sdílených dat závisí na tom, který proces skončí jako poslední.
  - Alternativně: když dvě různé vlákna přistupují ke stejným datům a nejsou kauzálně uspořádány a alespoň jeden přístup je zápis.

- Aby se zabránilo souběhu, musí být souběžné procesy synchronizovány.

(Deitel, Deitel & Choffness, 2004; Silberschatz, Galvin & Gagne, 2013)

Problém kritické sekce

- Procesy lze provádět souběžně
  - Můžou být kdykoli přerušeny
  - Vykonávají svůj kód po částech
- Souběžný přístup ke sdíleným datům může vést k nekonzistenci dat.
  - Potřeba zajistit konzistenci sdílených údajů.

- *n* procesů sdílí společnou vyrovnávací paměť pevné velikosti
- Každý proces má část kódu, nazvaný *kritická sekce*, ve kterém přistupuje ke sdílené paměti (*kritická oblast*).
- Problém
  - zajistit, aby v případě, že jeden proces vykonává svou kritickou sekci, nebyl žádný jiný proces ve své (související) kritické sekci.

(Silberschatz, Galvin & Gagne, 2013)

Problém kritické sekce

- Procesy používají a modifikují sdílená data

- Operace zápisu musí být vzájemně výlučné
- Operace zápisu musí být vzájemně výlučné s operacemi čtení
- Operace čtení (bez modifikace) mohou být souběžné
- Pro zabezpečení integrity dat se používají zámky

- Problém čtenářů a písařů

(Deitel, Deitel & Choffness, 2004; Silberschatz, Galvin & Gagne, 2013)

Příklad

- `counter++;`
- Může být implementováno jako

```
reg1 = counter          mov counter, %eax
reg1 = reg1 + 1         add 1, %eax
counter = reg1          mov %eax, counter
```

- `counter--;`
- Může být implementováno jako

```
reg2 = counter          mov counter, %eax
reg2 = reg2 - 1         sub 1, %eax
counter = reg2          mov %eax, counter
```

```
P1 provede      reg1 = counter          {reg1 = 5}
P1 provede      reg1 = reg1 + 1         {reg1 = 6}
P2 provede      reg2 = counter          {reg2 = 5}
P2 provede      reg2 = reg2 - 1         {reg2 = 4}
P1 provede      counter = reg1          {counter = 6 }
P2 provede      counter = reg2          {counter = 4}
```

Na konci muže být counter roven 6 nebo 4, ale správně je 5 (což se většinou podaří).
Chyba je důsledkem nepředvídatelného prokládání procesů/vláken vlivem preempce.

(Deitel, Deitel & Choffness, 2004; Silberschatz, Galvin & Gagne, 2013; Štěpán, 2018)

Řešení problému kritické sekce

- **Vzájemné vyloučení**. *Mutual Exclusion***.** Pokud se proces P_i nachází ve své kritické sekci, pak žádné další procesy nesmí být kritické sekci sdružené se stejným prostředkem.
- **Progres**. Pokud není v kritické sekci proces a existují nějaké procesy, které si přejí vstoupit do své kritické sekce, tak nelze výběr procesu pro vstup do kritické sekce odkládat na neurčito.
- **Konečné čekání.** Musí existovat omezení, kolikrát mohou ostatní procesy vstoupit do svých kritických sekcí poté, co proces podal žádost o vstup do své kritické sekce.
  - Podmínka spravedlivosti
  - Nesmíme dělat předpoklady o rychlosti a počtu CPU (případně procesů)

(Deitel, Deitel & Choffness, 2004; Silberschatz, Galvin & Gagne, 2013)

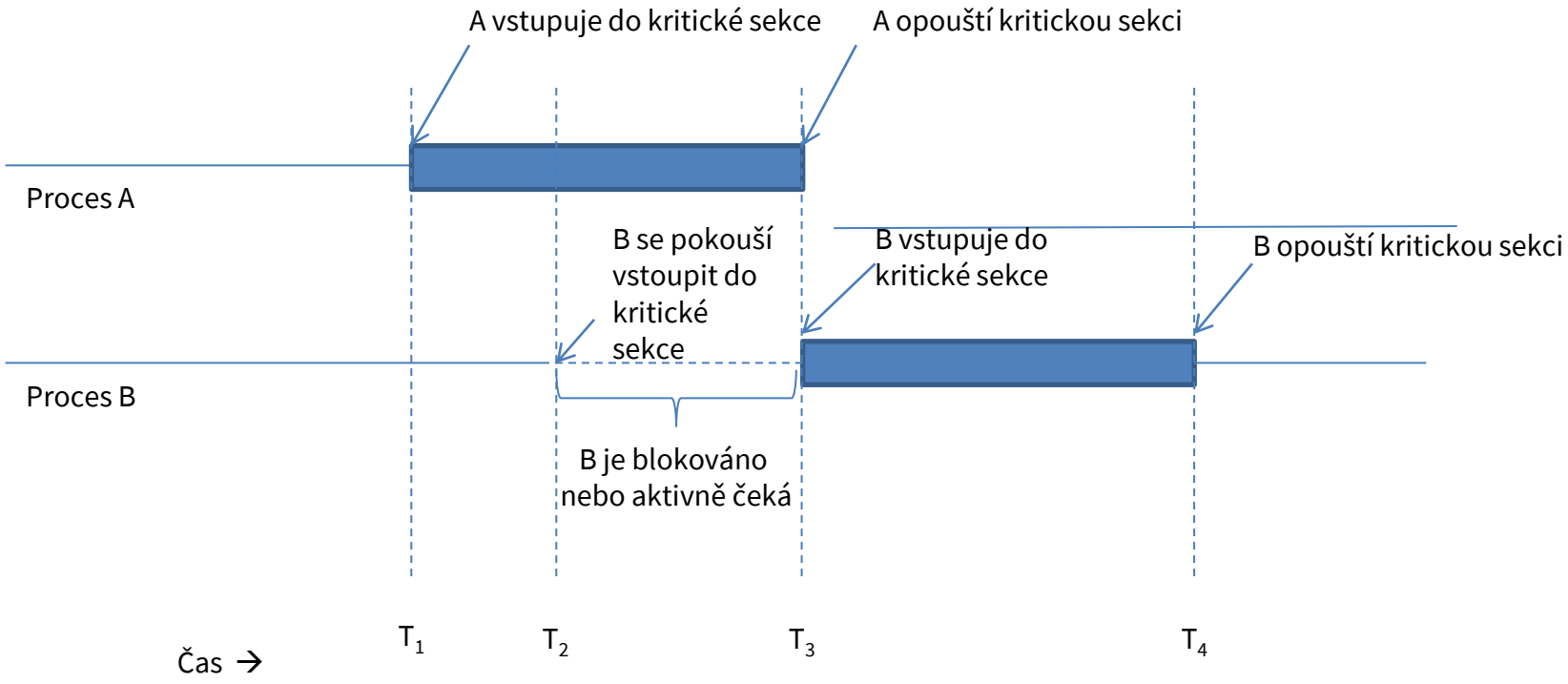
Předpoklady

- Ve své kritické sekci nesmí být současně dva procesy.
- Žádný proces by neměl čekat věčně, než vstoupí do své kritické sekce.
- Žádný proces běžící mimo kritickou sekci nemůže blokovat jiné procesy.
- Nesmíme dělat předpoklady o rychlosti a počtu CPU (případně procesů).

- Některé instrukce jsou *atomické* (nedělitelné)
  - jejich provedení je nepřerušitelné
    - Např. načíst, uložit, test instrukce
- Hardwarová konfigurace se může lišit
  - Jeden nebo více procesorů

Vzájemné vyloučení (*Mutual Exclusion)*

- Řešení je založeno na myšlence zamykání
  - Ochrana kritických oblastí pomocí zámků



(Deitel, Deitel & Choffness, 2004)

Aktivní čekání (Busy Waiting)

- Nepřetržité testování proměnné, dokud se neobjeví nějaká hodnota
  - Obvykle by se tomu snažíte vyhnout, protože plýtváte časem CPU.
- Zámek, který používá aktivní čekání.
  - Pouze pokud existuje důvodný předpoklad, že čekání bude krátké
  - Např. *spinlock*

```
while (turn != 1);
doSomething();
```

(Deitel, Deitel & Choffness, 2004)

Vzájemné vyloučení s aktivním čekáním
- Zákaz přerušení
- Lock proměnné
- Petersonovo řešení
- TSL Instrukce
- …

Možné řešení kritické sekce

- Software - aplikace
  - Algoritmy, které se nespoléhají na další podporu
  - Základní řešení, které je vždy s aktivním čekáním
- Hardware
  - Použijte speciální instrukce CPU (tsl, xchg,…)
  - Stále s aktivním čekáním
- Software - jádro
  - operační systémy poskytují systémová volání, která blokují proces (pasivní čekání)
    - např. **semafor, mutex**, …
  - Podpora synchronizačních služeb (např. monitory, zasílání zpráv)
  - Ostatní procesy mohou používat CPU, když je volající blokován.

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

(Lažanský, 2014; Silberschatz, Galvin & Gagne, 2013)

Softwarové aplikační řešení synchronizace

- Použití sdílené zamykací proměnné – např. *lock*
  - Před vstupem do kritické sekce proces testuje proměnnou
    - je-li nulová, nastaví hodnotu na 1 a vstoupí do kritické sekce
    - Jestliže není hodnota 0, proces čeká ve smyčce a testuje hodnotu
  - Při opouštění kritické sekce proces tuto proměnnou nuluje
- Vzájemné vyloučení s aktivním čekáním
  - Nic se nevyřešilo, jenom se možnost souběhu přenesla na sdílenou proměnnou *lock.*

(Lažanský, 2014; Silberschatz, Galvin & Gagne, 2013)

Softwarové aplikační řešení synchronizace

- Striktní střídání procesů pomocí proměnné *turn*
  - Hodnota *turn* určuje, který proces smí vstoupit do kritické sekce.
    - Je-li *turn=0*, tak vstoupí proces P0
    - Je-li *turn=1*, tak vstoupí proces P1
  - Při ukončení kritické sekce nastaví P0 na jedničku a P1 na nulu.

- Je porušen požadavek nepřípustné závislosti na rychlosti
  - Předpokládáme, že procesy jsou přibližně stejně rychlé a střídají se.
- Jestliže bude jeden proces výrazně rychlejší, tak bude muset čekat než druhý projde kritickou sekcí a nastaví hodnotu *turn*.
  - Porušen požadavek na progres.

(Lažanský, 2014; Silberschatz, Galvin & Gagne, 2013)

Hardwarové řešení synchronizace

- Přístup k zamykacím proměnným musí být atomický
- Jednoprocesorové systémy mohou vypnout přerušení
  - Pouze jádro, nelze použít na aplikační úrovni. Privilegovaná akce.
- Speciální atomické (nedělitelné) instrukce
  - Instrukce TestAndSet atomicky přečte obsah adresované buňky a změní její obsah (tsl)
  - Instrukce Swap (xchg) vymění obsah registru a adresované buňky
  - Prefix LOCK (P6 family CPU). Signál LOCK zajistí, že použití v systému s vice procesory má procesor výlučný přístup k jakékoliv sdílené paměti.

(Lažanský, 2014)

Mutex

- Mutex - *Mutual Exclusion*, vzájemné vyloučení
  - blokovací mechanismus používaný k synchronizaci přístupu ke zdroji.
- Chrání kritickou sekci nejprve získáním zámku *acquire()*
- Při opuštění kritické sekce uvolní zámek *release()*
  - Boolean proměnná. Udává, zda je zámek k dispozici nebo ne.
  - Volání na *acquire()* a *release()* musí být atomická
    - Obvykle je implementováno hardwarově pomocí atomických instrukcí
- Mutex může získat pouze jeden úkol (vlastník mutexu).

(Silberschatz, Galvin & Gagne, 2013)

```
do {
   acquire lock
            critical section
   release lock
   remainder section
} while (true);

acquire() {
     while (!available) ;
              /* busy wait */
     available = false;;
   }

  release() {
     available = true;
   }
```

Spinlock
(Silberschatz, Galvin & Gagne, 2013)

Pthread Synchronizace

- Pthreads API je nezávislé na OS
- Poskytuje:
  - Mutexové zámky
  - Podmínkové proměnné
- Nepřenosná rozšíření zahrnují:
  - zámky pro čtení a zápis
  - spinlocky

(Silberschatz, Galvin & Gagne, 2013)

Thread call	Description
pthread_mutex_init	Create a mutex
pthread_mutex_destroy	Destroy an existing mutex
pthread_mutex_lock	Acquire a lock or block
pthread_mutex_trylock	Acquire a lock or fail
pthread_mutex_unlock	Release a lock

Thread call	Description
pthread_cond_init	Create a condition variable
pthread_cond_destroy	Destroy a condition variable
pthread_cond_wait	Block waiting for a signal
pthread_cond_signal	Signal another thread and wake it up
pthread_cond_broadcast	Signal multiple threads and wake all of them

Semafor

- Semafor je zobecněný mutex. Je to celočíselná proměnná (integer)
- Lze k němu přistupovat pomocí dvou atomických operací.
- wait () nebo down () a signal () or up ()
  - Původně pojmenované **P ()** a **V ()** Dijkstra (1965) Proberen / Verhogen

- Definice operace down ()

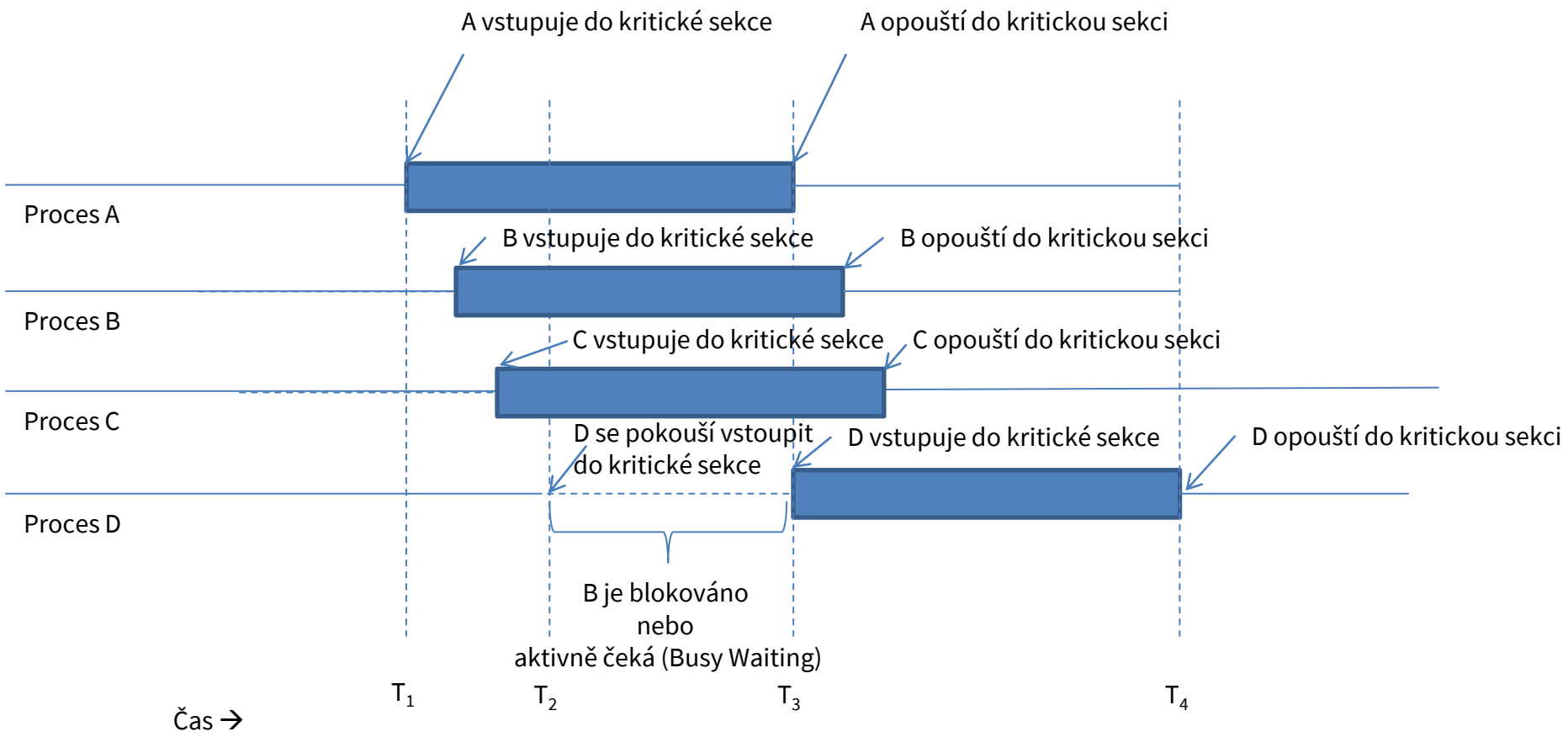
```
down (S) {
   while (S <= 0)
     ;     // blokování nebo aktivní čekání
   S--;
}
```

- Definice operace up ()

```
up(S) {
   S++;   // nebo vzbudit čekající proces
}
```

(Deitel, Deitel & Choffness, 2004; Silberschatz, Galvin & Gagne, 2013)

Semafor



(Deitel, Deitel & Choffness, 2004; Silberschatz, Galvin & Gagne, 2013)

Rozdíly mezi semaforem a mutexem

- Mutex je uzamykací mechanismus používaný k synchronizaci přístupu ke zdroji.
  - Pouze jeden úkol (vlákno nebo proces) může získat mutex.
    - S mutexem je spojeno vlastnictví a zámek může uvolnit pouze vlastník.
  - Umožňuje více úkolům přístup k jednomu prostředku, ale pouze jednotlivě.

- Semafor je signalizační mechanismus - signál „Jsem hotový, můžete pokračovat“
  - Hodnotu semaforu může změnit kterákoliv úloha pracující se zdrojem.
  - Umožňuje více úkolům přistupovat k určitému počtu instancí zdroje.

Spinlock

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

(Štěpán, 2018; Lažanský, 2014)

Synchronizace v Linuxu

- Před jádrem verze 2.6 byla krátké kritická sekce implementována pomocí zákazu přerušení.
- Od verze 2.6, plně preemptivní
- Linux poskytuje:
  - semafory
  - spinlocky
  - Reader/writer verze semaforů a spinlocků
- V systému s jedním procesorem je spinlock nahrazen zákazem preempce jádra

(Silberschatz, Galvin & Gagne, 2013)

POSIX Semaphore

- *sem_init(sem_t *sem, int pshared, unsigned int value);*
  - Semafor je inicializován voláním sem_init(pro procesy nebo vlákna) nebo sem_open (pro IPC).
    - *sem* : Specifies the semaphore to be initialized.
    - *pshared* : This argument specifies whether or not the newly initialized semaphore is shared between processes or between threads.
      - A non-zero value means the semaphore is shared between processes and a value of zero means it is shared between threads.
    - *value* : Specifies the value to assign to the newly initialized semaphore.

- *int sem_wait(sem_t *sem);*
  - Locks a semaphore or wait
- *int sem_post(sem_t *sem);*
  - Releases or signal a semaphore
- *int sem_getvalue(sem_t *sem, int *valp);*
  - *Gets the current value of semaphor and places it in the location pointed to by valp*
- *sem_destoy(sem_t *sem);*
  - Destroys a semaphore

POSIX Semaphore

- *Vysvětlení kódu*
- Vytvoření dvou vláken
  - Druhé 2 vteřiny po prvním
- První vlákno spí 4 vteřiny po získání zámku
- Takže druhé vlákno nemůže vstoupit ihned
  - Vstoupí po 4 – 2 = 2 vteřinách

(SHANDILYA , 2018)

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

sem_t mutex;

void* thread(void* arg)
{
    //wait
    sem_wait(&mutex);
    printf("\nEntered..\n");

    //critical section
    sleep(4);

    //signal
    printf("\nJust Exiting...\n");
    sem_post(&mutex);
}


int main()
{
    sem_init(&mutex, 0, 1);
    pthread_t t1,t2;
    pthread_create(&t1,NULL,thread,NULL);
    sleep(2);
    pthread_create(&t2,NULL,thread,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    sem_destroy(&mutex);
    return 0;
}
```

Monitory

- Vysokoúrovňová abstrakce, která poskytuje pohodlný a účinný mechanismus pro synchronizaci procesů.
- Speciální konstrukce programovacího jazyka
  - Abstraktní datový typ
  - Vnitřní proměnné přístupné pouze kódem v rámci funkce.
  - Implementace monitoru je systémově závislá a využívá prostředků JOS
    - Obvykle semaforů
- Současně může být na monitoru aktivní pouze jeden proces
  - Procedury definované jako monitorové procedury se vždy vzájemně vylučují
  - Procesy, které chtějí vykonávat monitorovou proceduru, jsou řazeny do fronty

(Silberschatz, Galvin & Gagne, 2013; Štěpán, 2018; Lažanský, 2014)

Negativní důsledky synchronizace

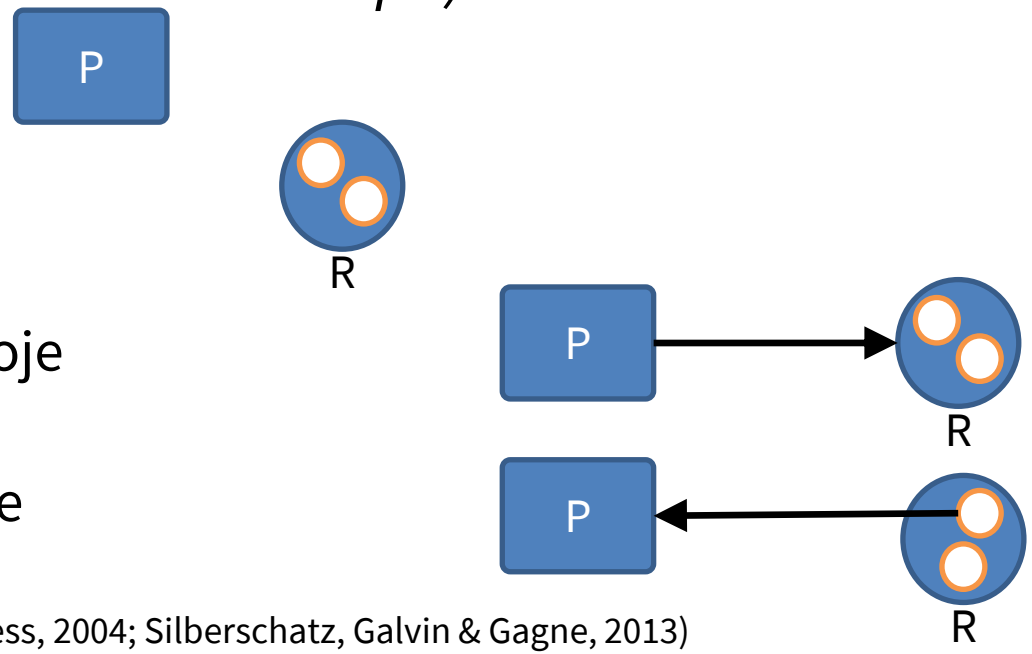
- Zablokování procesu může způsobit
  - Uváznutí (deadlock)
    - Dva nebo více procesů čekají na událost (zprávu, prostředek), ke které by mohlo dojít jedině v tom případě, že by jeden z těchto procesů pokračoval.
  - Vyhladovění (starvation)
    - Dva procesy se střídají ve své kritické sekci a zamezí tak přístupu třetímu.
  - Aktivní zablokování (livelock)
    - Procesy se snaží aktivně řešit uváznutí, bohužel neúspěšně.
  - Inverze priorit (priority inversion)
    - Proces P_0 s nízkou prioritou vlastní prostředek požadovaný procesem P_3 s vysokou prioritou, což proces P_3 zablokuje. Proces P_2 se střední prioritou, který sdílený prostředek nepotřebuje, poběží stále a nedovolí tak procesu P_0 prostředek uvolnit.

(Lažanský, 2014)

Uváznutí (Deadlock)

- Posloupnost událostí potřebných k použití zdroje:
  - Vyžádání zdroje
  - Použití zdroje
  - Uvolnění zdroje

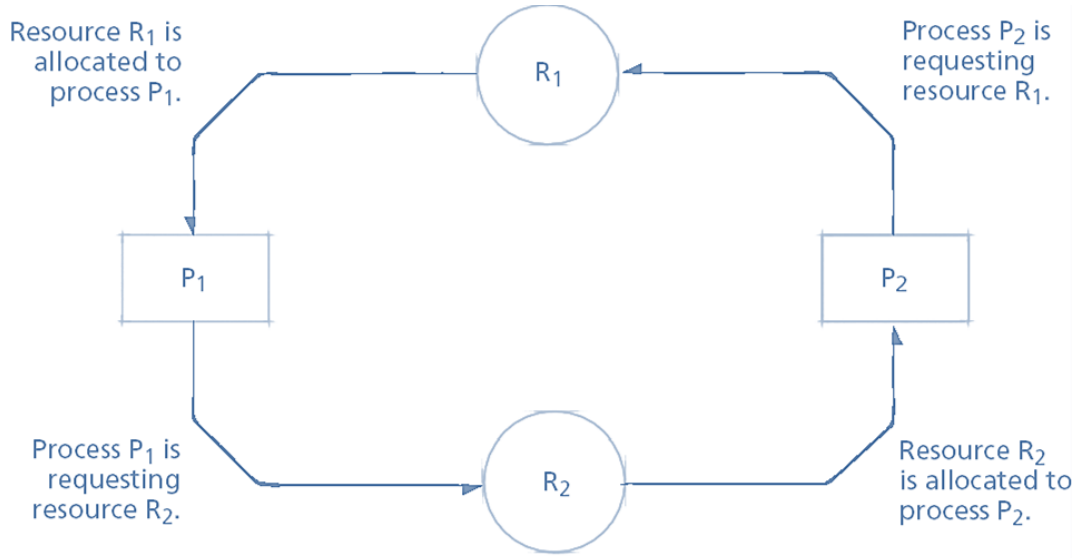
- Graf přidělování zdrojů (*Resource Allocation Graph)*
  - Proces
  - Zdroj se 2 instancemi
  - Požadavek procesu na instanci zdroje
  - Proces drží (používá) instanci zdroje



(Deitel, Deitel & Choffness, 2004; Silberschatz, Galvin & Gagne, 2013)

Uváznutí (Deadlock)

- Jednoduché uváznutí
  - Tento systém uváznul, protože každý proces drží prostředek požadovaný jiným procesem a žádný proces není ochoten uvolnit prostředek, který drží.



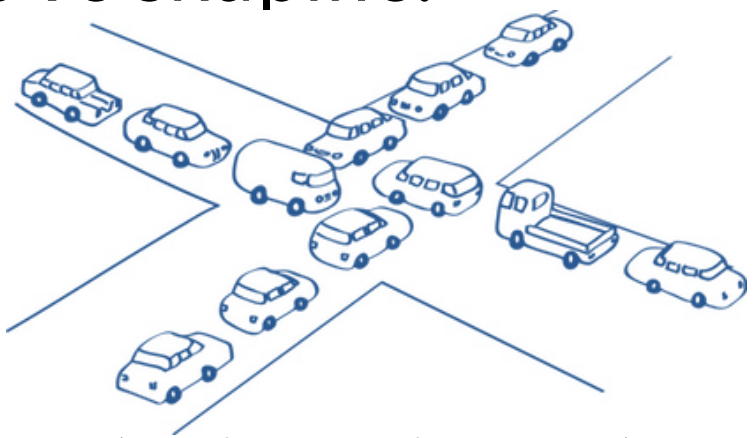
(Deitel ,Deitel & Choffness, 2004)

Uváznutí (Deadlock)

- Proces nebo vlákno čeká na určitou událost, která nenastane

(Deitel, Deitel & Choffness, 2004)

- Skupina procesů je zablokována, pokud každý proces ve skupině čeká na událost, kterou může způsobit pouze jiný proces ve skupině.

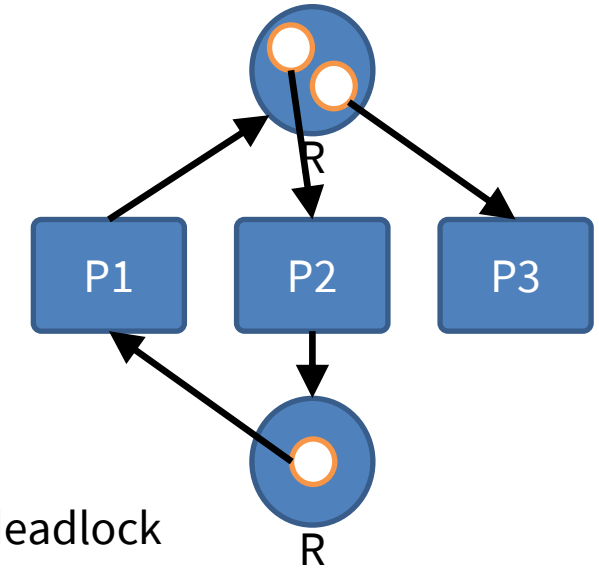


(Source: https://copycode.tistory.com/78)

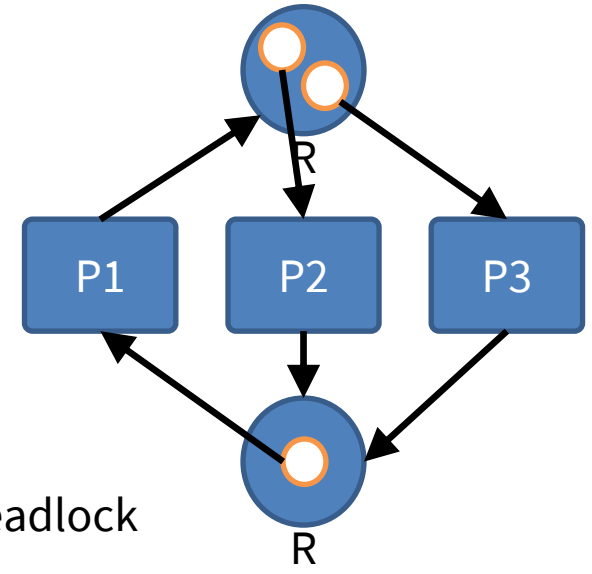
(Tanenbaum, 2015)

(Source: https://allstarnix.blogspot.com/2012/07/real-life-deadlock.html)

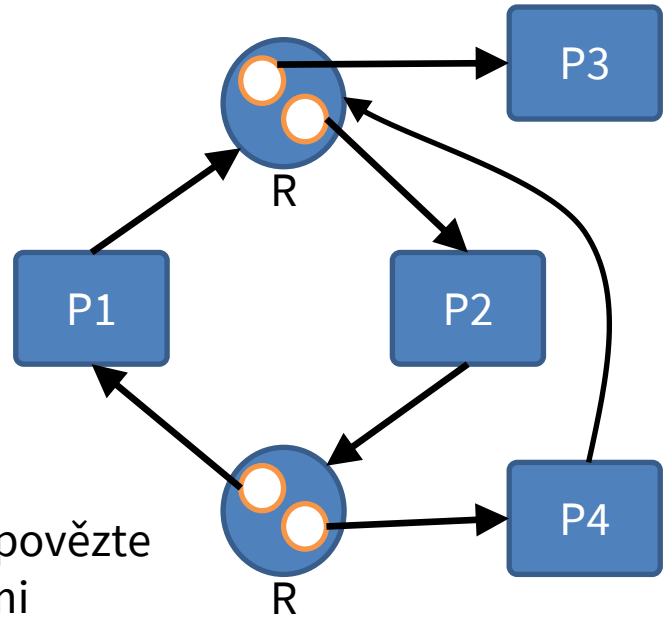
Graf přidělování zdrojů



Není deadlock



Deadlock



Odpovězte sami

Podmínky uváznutí (Coffman's Conditions)

- *Uváznutí může nastat*, pouze pokud budou současně splněny všechny čtyři podmínky.

- Podmínka vzájemného vyloučení (Mutual Exclusion)
  - Zdroj může používat v jednom okamžiku pouze jeden proces. Nesdílitelnost zdrojů.
- Postupné uplatňování požadavků (Hold&Wait)
  - Proces drží přidělené zdroje a čeká na uvolnění dalších zdrojů, aby je získal.
- Bez preempce
  - Přidělené zdroje nejdou násilně odebrat.
- Zacyklení požadavků, cyklické čekání (Circular-wait)
  - Dva nebo více procesů čeká v „kruhovém řetězci" na jeden nebo více zdrojů, které drží další proces v řetězci.
    - V případě zdrojů s jednou instancí je tato podmínka postačující

(Deitel, Deitel & Choffness, 2004; Silberschatz, Galvin & Gagne, 2013)

Řešení uváznutí

- Prevence uváznutí (*Deadlock Prevention*)
  - Zajištění, že uváznutí nemůže nikdy nastat (Porušením podmínek vzniku).
- Vyhýbání se uváznutí (*Deadlock Avoidance*)
  - Zajištění, že nikdy nenastane možnost vzniku uváznutí (bezpečný stav).
    - Pomocí opatrné alokace zdrojů. Při hrozbě vzniku uváznutí není prostředek přidělen.
      - Nebezpečí vzniku vyhladovění.
- Detekce uváznutí a obnova (*Deadlock detection and recovery*)
  - Uváznutí může nastat, ale je detekováno a zajistí se obnova stavu před uváznutím.
- Ignorování hrozby (*Ignoring deadlock)*

(Deitel, Deitel & Choffness, 2004; Silberschatz, Galvin & Gagne, 2013; Tanenbaum, 2015)

Prevence uváznutí

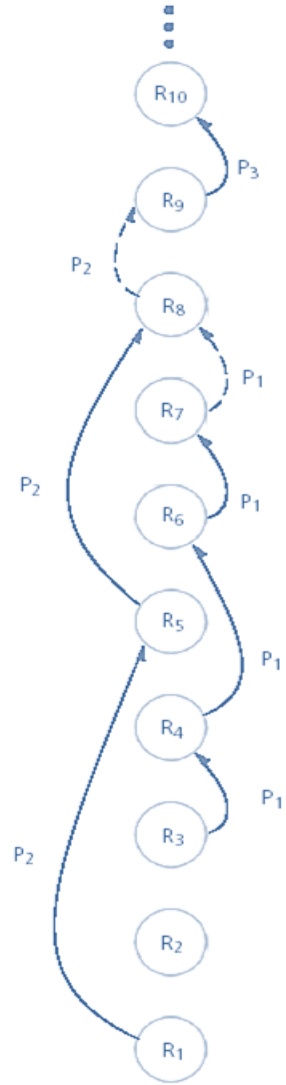
- K uváznutí nemůže dojít, pokud je porušena alespoň jedna ze čtyř podmínek.

- Vzájemné vyloučení
  - Není požadováno pro sdílitelné prostředky (např. soubory pouze pro čtení), bohužel musí platit pro nesdílitelné prostředky.
  - Virtualizace (Spooling) prostředků
- Postupné uplatňování požadavků(Hold&Wait)
  - musíme zaručit, že kdykoli proces požaduje zdroj, nemá žádné další zdroje
    - Např. Proces požaduje všechny prostředky na začátku nebo před alokací dalších musí všechny dosavadní vrátit a alokovat je znovu najednou.
      - Problémem je nízké využití zdrojů a možnost vzniku vyhladovění.

(Silberschatz, Galvin & Gagne, 2013; Tanenbaum, 2015; Deitel, Deitel & Choffness, 2004; Štěpán, 2018; Lažanský, 2014)

Prevence uváznutí

- Porušení podmínky „Bez preempce“
  - Povolení násilného odebírání zdrojů je velmi riskantní (ztráta stavu)
  - Přesto některé zdroje mohou být násilně odebírány
    - Dokážeme uchovat jejich stav a následně se vrátit do stavu před přerušením
- Zabránění zacyklení požadavků na zdroje
  - Zdroje jsou očíslovány a procesy je mohou alokovat pouze ve vzrůstajícím pořadí čísel zdrojů.
    - Problematické, protože zdroje vznikají a zanikají dynamicky.



Linear ordering of resources (Deitel, 2004)

(Silberschatz, Galvin & Gagne, 2013; Tanenbaum, 2015; Deitel, Deitel & Choffness, 2004; Štěpán, 2018; Lažanský, 2014)

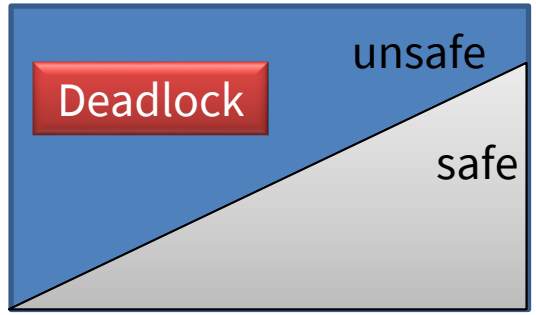
Vyhýbání se uváznutí (Deadlock Avoidance)

- Vyžaduje nějaké další a priori informace
  - Např. každý proces musí deklarovat maximální počet zdrojů každého typu, které bude potřebovat.
    - Nejjednodušší a nejužitečnější model.
    - Algoritmus bankéře Dijkstra
- Dynamicky zkoumá stav přidělování prostředků, aby se zajistilo, že nikdy nebude splněna podmínka cyklického čekání.
  - Stav přidělování prostředků je definován počtem dostupných a přidělených zdrojů a maximálním požadavkem procesů na prostředky
    - Stav může být bezpečný nebo nebezpečný (nesmí se do něj dostat, protože hrozí uváznutí)

(Silberschatz, Galvin & Gagne, 2013)

Dijkstrův Bankeřův algoritmus (Deadlock Avoidance)

- Když proces požaduje dostupný zdroj, systém musí rozhodnout, zda okamžité přidělení zanechá systém v bezpečném stavu
  - Bezpečný stav
    - OS může zaručit, že všechny procesy dokážou dokončit svou práci v konečném čase.
  - Nebezpečný stav
    - OS nemůže zaručit, že všechny procesy dokážou dokončit práci v konečném čase.
      - Neznamená to, že nastalo uváznutí
        - » Murphyho zákony: Uváznutí nastane vždy, když to nejméně čekáte



(Silberschatz, Galvin & Gagne, 2013)

Příklad bankéřova algoritmu

Proces	MAX (maximum potřebných)	Alokované zdroje	Požadované zdroje
P1	4	1	3
P2	6	4	2
P3	8	5	3
Celkový počet zdrojů: 12		Dostupné zdroje: 2	

Bezpečný stav (Safe state)

Proces	MAX (maximum potřebných)	Alokované zdroje	Požadované zdroje
P1	10	8	2
P2	5	2	3
P3	3	1	2
Celkový počet zdrojů: 12		Dostupné zdroje: 1	

Nebezpečný stav (Unsafe state)

(Deitel, Deitel & Choffness, 2004)

Příklad bankéřova algoritmu

Proces	MAX (maximum potřebných)	Alokované zdroje	Požadované zdroje
P1	5	1	4
P2	3	1	2
P3	10	5	5
Celkový počet zdrojů: 10		Dostupné zdroje: 3	

Odpovězte sami

Proces	MAX (maximum potřebných)	Alokované zdroje	Požadované zdroje
P1	10	8	2
P2	5	1	4
P3	3	1	6
Celkový počet zdrojů: 12		Dostupné zdroje: 2	

Odpovězte sami

(Deitel, Deitel & Choffness, 2004)

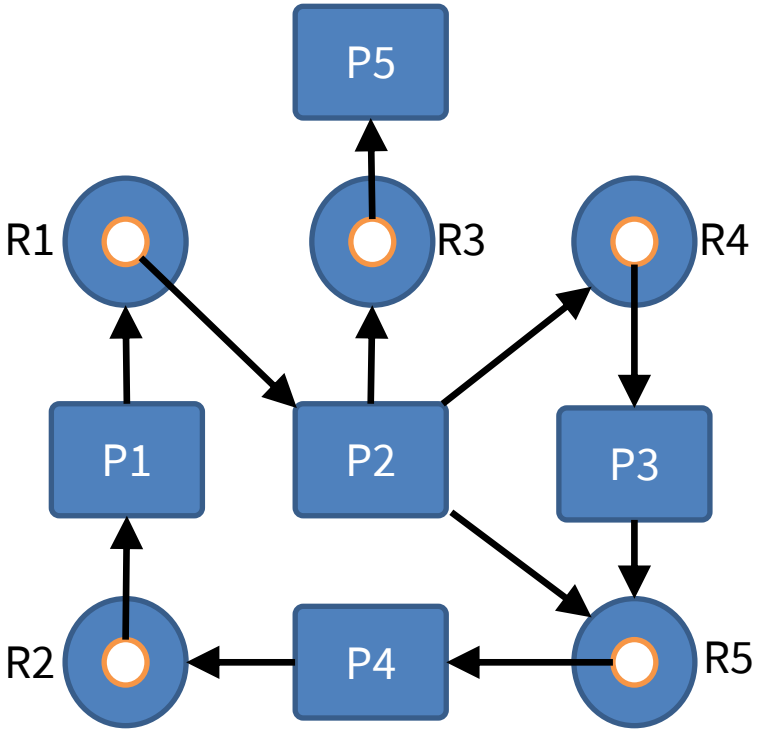
Detekce uváznutí a obnova

- Systému může vstoupit do nebezpečného stavu a připouští vznik uváznutí.
- Algoritmus detekce
  - Udržujte čekací graf (Wait-for Graph) - uzly jsou procesy.
  - Pravidelně vyhledává cyklus v grafu. Pokud existuje cyklus, existuje uváznutí.
- Obnova systému
  - Zotavení pomocí preempce zdrojů - výběr oběti (minimalizovat náklady)
  - Obnova prostřednictvím návratu do předchozího bezpečného stavu (rollback a restart procesu)
  - Zotavení pomocí zabíjení procesů
    - Zruší všechny uvázlé procesy nebo jeden po druhém, dokud nebude odstraněno zacyklení požadavků
    - V jakém pořadí bychom se měli rozhodnout pro zrušení procesů?
      - Priorita procesu, doba běhu procesu a za jak dlouho se má dokončit, Zdroje, které proces použil. Kolik procesů bude třeba ukončit, Je proces interaktivní nebo dávkový?

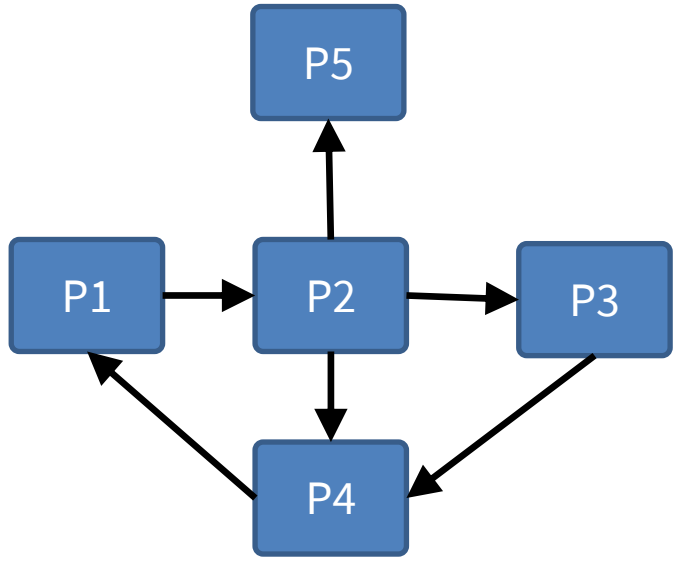
(Deitel, Deitel & Choffness, 2004; Silberschatz, Galvin & Gagne, 2013; Tanenbaum, 2015)

Detekce uváznutí

Graf přidělování zdrojů
(*Resource Allocation Graph*)



Čekací graf
(Wait-for Graph)



(Silberschatz, Galvin & Gagne, 2013)

Livelock

- Livelock je riziko, kdy se procesy snaží aktivně řešit uváznutí, bohužel neúspěšně.
- Livelock je zvláštní případ vyhladovění (resource starvation)
- Příkladem reálného světa v reálném světě je situace, kdy se dva lidé setkávají v úzké chodbě a každý se snaží zdvořile ustoupit stranou a nechat druhého projít. Bohužel obě strany ustupují na stejnou stranu ve stejný čas a pořád stojí před sebou.

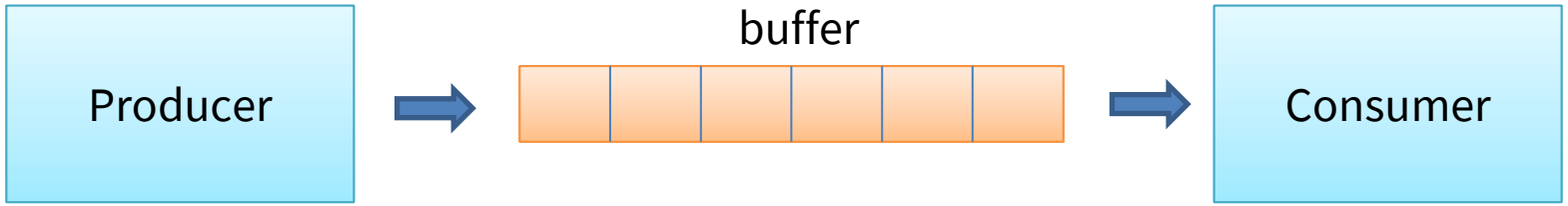
Příklady IPC problémů

- Literatura o operačních systémech je plná zajímavých problémů
- Jsou široce diskutované a využívající různé metody synchronizace.

- Producent-konzument
- Stolující filozofové
- Čtenáři a písaři
- Spící kadeřník
- …

Producent-konzument

- Sdílená vyrovnávací paměť pevné velikosti
- Proměnná *counter* udržuje počet položek ve vyrovnávací paměti.
- Producent vkládá položky do sdílené vyrovnávací paměti a inkrementuje počítadlo *counter.*
- Konzument odebírá položky ze sdílené vyrovnávací paměti a snižuje hodnotu počítadla *counter*.



Producent-konzument (špatné řešení)

```
# define N 100;                                    //number of slots in the buffer
int counter = 0;                                   //number of items in the buffer

void producer(void)
{
    int item;
    while (true) {                                 //repeat forever
        item = produce_item();                     //generate next item
        if (counter == N) sleep();                 //if buffer is full, go to sleep
        insert_item();                             //put item in the buffer
        counter++;                                 //increment count of items in the buffer
        if (counter == 1) wakeup(consumer);        //if buffer was empty, wake up consumer
    }
}

void consumer(void)
{
    int item;
    while (true) {                                 //repeat forever
        if (counter == 0) sleep();                 //if buffer is empty, go to sleep
        item = remove_item();                      //take item out of the buffer
        counter--;                                 //decrement count of items in buffer
        if (counter == N-1) wakeup(producer);      //if buffer was full, wake up producer
        Consume_item();                            //do something with item
    }
}
```

(Tanenbaum, 2015)

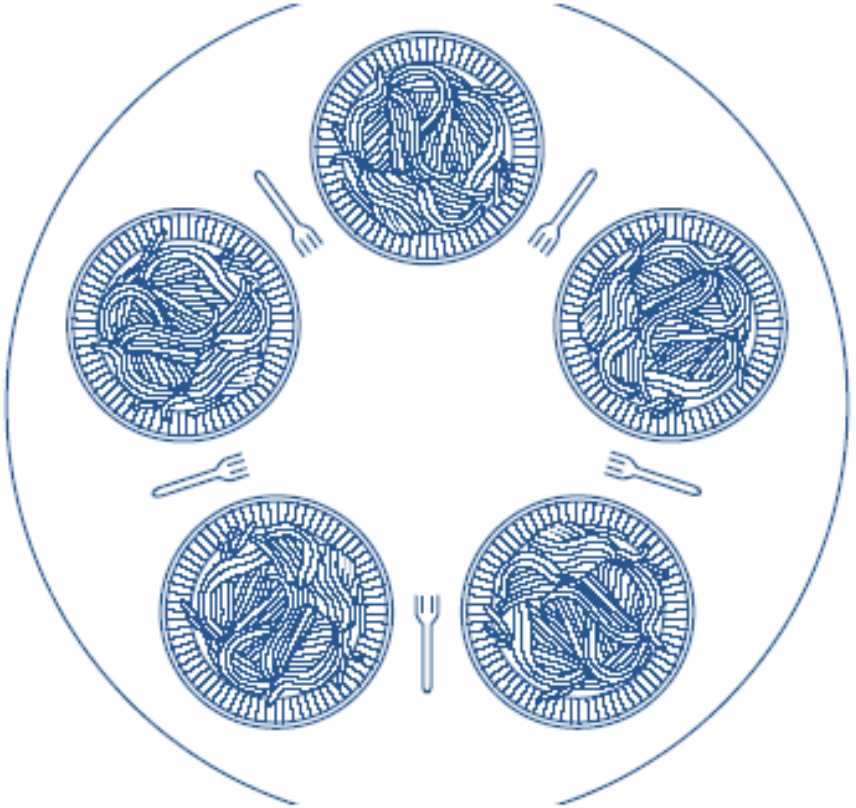
Producent-konzument

```
#define N 100                          // number of slots in the buffer
typedef int semaphore;                 /* semaphores are a special kind of int */
semaphore mutex = 1;                   /* controls access to critical region */
semaphore empty = N;                   /* counts empty buffer slots */
semaphore full = 0;                    /* counts full buffer slots */
void producer(void)
{
    int item;
    while (TRUE) {                     /* TRUE is the constant 1 */
        item = produce_item();         /* generate something to put in buffer */
        down(&empty);                  /* decrement empty count */
        down(&mutex);                  /* enter critical region */
        insert_item(item);             /* put new item in buffer */
        up(&mutex);                    /* leave critical region */
        up(&full);                     /* increment count of full slots */
    }
}
void consumer(void)
{
    int item;
    while (TRUE) {                     /* infinite loop */
        down(&full);                   /* decrement full count */
        down(&mutex);                  /* enter critical region */
        item = remove_item();          /* take item from buffer */
        up(&mutex);                    /* leave critical region */
        up(&empty);                    /* increment count of empty slots */
        consume_item(item);            /* do something with the item */
    }
}
```

(Tanenbaum, 2015)

Stolující filozofové

- Kolem kruhového stolu sedí pět filozofů.
- Každý střídá jídlo a přemýšlení
- Před každým filosofem je mísa špaget, která je neustále doplňována.
- Na stole je přesně pět vidliček tak, že leží mezi filozofy.
- Špagety vyžadují, aby filozof používal obě dvě vidličky současně.



(Tanenbaum, 2015)

Stolující filozofové (špatné řešení)

```
#define N 5                                     // number of philosophers
void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think();                                /* philosopher is thinking */
        take_fork(i);                           /* take left fork */
        take_fork((i+1) % N);                   /* take right fork */
        eat();                                  /* yum-yum, spaghetti */
        put_fork(i);                            /* put left fork back on the table */
        put_fork((i+1) % N);                    /* put right fork back on the table */
    }
}
```

(Tanenbaum, 2015)

Stolující filozofové (1)

```
#define N 5                     // number of philosophers
#define LEFT (i+N-1)%N          // number of i's left neighbor
#define RIGHT (i+1)%N           // number of i's right neighbor
#define THINKING 0              // philosopher is thinking
#define HUNGRY 1                // philosopher is trying to get forks
#define EATING 2                // philosopher is eating

typedef int semaphore;          /* semaphores are a special kind of int */
int state[N];                   /* array to keep track of everyone's state */
semaphore mutex = 1;            /* mutual exclusion for critical regions */
semaphore s[N];                 /* one semaphore per philosopher */

void philosopher(int i)         /* i: philosopher number, from 0 to N1 */
{
    while (TRUE){               /* repeat forever */
        think();                /* philosopher is thinking */
        take_forks(i);          /* acquire two forks or block */
        eat();                  /* yum-yum, spaghetti */
        put_forks(i);           /* put both forks back on table */
    }
}
```

(Tanenbaum, 2015)

Stolující filozofové (2)

```
void take_forks(int i)          /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);               /* enter critical region */
    state[i] = HUNGRY;          /* record fact that philosopher i is hungry */
    test(i);                    /* try to acquire 2 forks */
    up(&mutex);                 /* exit critical region */
    down(&s[i]);                /* block if forks were not acquired */
}

void put_forks(i)               /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);               /* enter critical region */
    state[i] = THINKING;        /* philosopher has finished eating */
    test(LEFT);                 /* see if left neighbor can now eat */
    test(RIGHT);                /* see if right neighbor can now eat */
    up(&mutex);                 /* exit critical region */
}

void test(i)                    /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

(Tanenbaum, 2015)

Čtenáři a písaři

- Několik procesů přistupuje ke společným datům
  - Některé procesy data jen čtou – čtenáři
  - Jiné procesy potřebují data zapisovat – písaři
- Libovolný počet čtenářů může přistupovat souběžně
  - Sdílí zdroj – mohou číst současně
- V jednom okamžiku smí daný zdroj modifikovat nejvýše jeden písař
  - Operace zápisu musí být exklusivní, vzájemně vyloučená s jakoukoli jinou operací (zápisovou i čtecí)
    - Jestliže písař modifikuje zdroj, nesmí ho současně číst žádný čtenář

(Tanenbaum, 2015; Lažanský, 2014)

Čtenáři a písaři

- Dva možné přístupy
  - Přednost čtenářů
    - Žádný čtenář nebude muset čekat, pokud sdílený zdroj nebude obsazen písařem. Jinak řečeno: Kterýkoliv čtenář čeká pouze na opuštění kritické sekce písařem.
    - Písaři mohou stárnout
  - Přednost písařů
    - Jakmile je některý písař připraven vstoupit do kritické sekce, čeká jen na její uvolnění (čtenářem nebo písařem). Jinak řečeno: Připravený písař předbíhá všechny připravené čtenáře.
    - Čtenáři mohou stárnout

(Lažanský, 2014)

Použitá a doporučená literatura

- DUARTE, Gustavo. Anatomy of a Program in Memory. *Many But Finite: Tech and science for curious people.* [online]. 2009, Jan 27th, 2009 [cit. 2018-02-28]. Dostupné z: https://manybutfinite.com/post/anatomy-of-a-program-in-memory/
- DEITEL H. M., DEITEL P. J. & CHOFFNES D. R.: *Operating systems*. 3rd ed., Pearson/Prentice Hall, 2004. ISBN 0131246968.
- TANENBAUM A. S.: *Modern operating systems*. 4th ed. Boston: Pearson, 2015. ISBN 0-13-359162-x.
- SILBERSCHATZ A., GALVIN P. B. & GAGNE G.: *Operating system concepts*. 9th ed. Hoboken, NJ: Wiley, 2013. ISBN 978-1-118-06333-0.
- STALLINGS W.: *Operating Systems: Internals and Design Principles.* 8th ed., Pearson Education Limited, 2014.

Použitá a doporučená literatura

- YOSIFOVICH, P., IONESCU, A., RUSSINOVICH, M.E., SOLOMON, D. A.: Windows Internals, Part 1: System architecture, processes, threads, memory management, and more (7th Edition). Microsoft Press, 2017.
- Yu-Hsin Hung. *Linux Kernel: Process Scheduling* [online]. Mar 25, 2016 [cit. 2019-02-07]. Dostupné z: https://medium.com/hungys-blog/linux-kernel-process-scheduling-8ce05939fabd
- HOFFMAN, Chris. What Is a “Zombie Process” on Linux?. *How-To Geek* [online]. September 28, 2016 [cit. 2019-02-07]. Dostupné z: https://www.howtogeek.com/119815/htg-explains-what-is-a-zombie-process-on-linux/
- Singh, Jaswinder Pal: Operating systems. Princeton University, 2018. [online]. Dostupné z: http://www.cs.princeton.edu/courses/cs318/

Použitá a doporučená literatura

- LAŽANSKÝ J.: Operační systémy a databáze. Přednášky A3B33OSD FELK ČVUT, [on-line], 2014.
- Štěpán P.: Operační systémy. Přednášky FEL ČVUT, [on-line], 2017.
- SHANDILYA P.: How to use POSIX semaphores in C language. [on-line], 2018, Dostupné z: https://www.geeksforgeeks.org/use-posix-semaphores-c/