

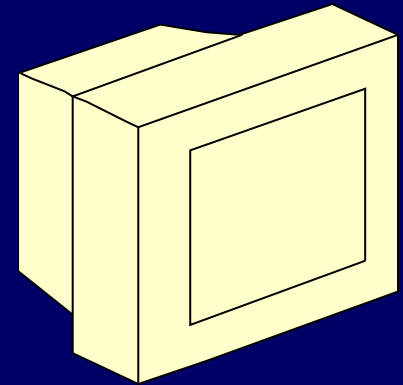
# 05. Meziprocesová komunikace – zprávy, RPC

ZOS 2012

---



# Monitory – opakování



- Kolik procesů může být najednou v monitoru?
- Kolik **aktivních** procesů může být najednou v monitoru?
- Co je to podmínková proměnná?
- Čím se liší následující sémantiky volání signal?
  - Hoare
  - Hansen
  - Java
- Musím uvnitř monitoru dávat pozor na současný přístup k proměnným monitoru?



# Semafor

## obecný koncept, programovací jazyk

### obecný koncept

semafor  
 $s = 0, 1, 2, \dots$   
 $P(), V()$

### Java

```
java.util.concurrent  
Semaphore (int ..)  
• acquire()  $\leftrightarrow$  P()  
• release()  $\leftrightarrow$  V()  
• tryacquire()
```

### C

```
#include<semaphore.h>  
sem.wait()  $\leftrightarrow$  P()  
sem.post()  $\leftrightarrow$  V()
```



---

# Meziprocesová komunikace

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



---

# Meziprocesová komunikace

Procesy mohou komunikovat:

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



# Linux - signály

Signály představují jednu z forem meziprocesové komunikace

Signál – speciální **zpráva** zaslaná procesu

(neobsahuje jinou informaci než číslo signálu)

Jsou **asynchronní** - mohou přijít kdykoliv, ihned jej proces obslouží

Signál je specifikován svým číslem, často se používají symbolická jména (SIGTERM, SIGKILL, ...)

Používají s v Linuxu, nejsou ve Windows v této podobě

**kill -9 1234** – pošle signál 9 procesu s PID: 1234

kill -l - vypíše signály



# Linux - signály

man 7 signal

## Události **generující** signály

- Stisk kláves (*CTRL+C* generuje *SIGINT*)
- HW přerušení (dělení nulou)
- Příkaz kill (1), systémové volání kill (2)
- Mohou je generovat uživatelské programy – kill (2)

## **Reakce** na signály

- Ignorování signálu (ale např. *SIGKILL*, *SIGSTOP* nelze)
  - Vlastní zpracování naší funkcí
  - Standardní zpracování
-

```
#!/bin/bash
obsluha() {
    echo "Koncim..."
    exit 0
}
# pri zachyceni signalu SIGINT se vykona fce obsluha
trap obsluha INT
```

```
SEC=0
while true ; do
    sleep 1
    SEC=$((SEC+1))
    echo "Jsem PID $$, ziju $SEC"
done
# sem nikdy nedojdeme
exit 0
```

Skript zareaguje na **Ctrl+C** zavoláním funkce **obluha()**  
Příkaz **trap** definuje jaká funkce se pro obsluhu daného signálu zavolá





# Linux – využití signálu při ukončení práce OS

Vypnutí počítače:

*INIT: Sending all processes the **TERM** signal*

*INIT: Sending all processes the **KILL** signal*

Proces init pošle všem podřízeným signál TERM

=> tím žádá procesy o ukončení a dává jim čas učinit tak korektně

Po nějaké době pošle signál KILL, který nelze ignorovat a způsobí ukončení procesu.

---



# Přehled vybraných signálů

<b>SIGTERM</b>	Žádost o ukončení procesu
SIGSEGV	Porušení segmentace paměti
SIGABORT	Přerušování procesu
SIGHUP	Odříznutí od terminálu (nohup ignoru.)
<b>SIGKILL</b>	Bezpodmínečné zrušení procesu
SIQQUIT	Ukončení terminálové relace procesu
SIGINT	Přerušování terminálu (CTRL+C)
SIGILL	Neplatná instrukce
SIGCONT	Navrácení z pozastavení procesu
<b>SIGSTOP</b>	Pozastavení procesu (CTRL+Z)
SIGINT	Ukončení procesu na popředí



# Problém sdílené paměti

- Vyžaduje **umístění** objektu ve sdílené paměti
  - **Někdy není vhodné**
    - **Bezpečnost** – globální data přístupná kterémukoliv procesu bez ohledu na semafor
  - **Někdy není možné**
    - Procesy běží na různých strojích, komunikují spolu po síti
  - Řešení – předávání zpráv
-



# Předávání zpráv – **send, receive**

Zavedeme 2 primitiva:

- **send** (**adresát**, zpráva) - odeslání zprávy
- **receive**(**odesilatel**, zpráva) - příjem zprávy

## □ Send

- Zpráva (libovolný datový objekt) bude zaslána adresátovi

## □ Receive

- Příjem zprávy od určeného odesilatele
  - Přijatá zpráva se uloží do proměnné (dat.struktury) „zpráva“
-

# Jak Linux?

```
mc - ~/zos2012/vlaknaC
MSGOP(2) Linux Programmer's Manual MSGOP(2)
NAME
    msgrcv, msgsnd - message operations
SYNOPSIS
    #include <sys/types.h>
    #include <sys/ipc.h>
    #include <sys/msg.h>

    int msgsnd(int msgid, const void *msgp, size_t msgsz, int msgflg);
    ssize_t msgrcv(int msgid, void *msgp, size_t msgsz, long msgtyp,
                  int msgflg);
DESCRIPTION
    The msgsnd() and msgrcv() system calls are used, respectively, to send
    messages to, and receive messages from, a message queue. The calling
    process must have write permission on the message queue in order to
    send a message, and read permission to receive a message.

    The msgp argument is a pointer to caller-defined structure of the folâ
    lowing general form:
Manual page msgsnd(2) line 1
```



# Vlastnosti

- synchronizace (blokující – neblokující)
- unicast, multicast, broadcast
- přímá komunikace x nepřímá komunikace
- délka fronty zpráv
- pevná vs. proměnná délka zprávy

Primitiva send a receive mohou mít celou řadu rozličných vlastností. V dalších slidech budou postupně rozebrány



# Synchronizace

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



# Send - receive

## □ **blokující send**

- čeká na převzetí správy příjemcem

## □ **neblokující send**

- vrací se ihned po odeslání zprávy
- většina systémů



## □ **blokující receive**

- není-li ve frontě žádná zpráva, zablokuje se
- většina systémů



## □ **neblokující receive**

- není-li zpráva, vrací chybu





# Receive s omezeným čekáním

- **receive (odesílatel, zpráva, t)**
  - čeká na příchod zprávy dobu  $t$
  - pokud zpráva nepřijde, vrací se volání s chybou

Další možná varianta



---

# Adresování

- send 1 příjemce nebo skupina?
    - Pošleme jednomu nebo více příjemcům
  
  - receive 1 odesílatel nebo různí?
    - Přijmeme pouze od jednoho odesílatele nebo od kohokoliv
-



# Skupinové a všesměrové adresování

- skupinové adresování (multicast)
    - zprávu pošleme **skupině** procesů
    - zprávu obdrží **každý** proces ve skupině
  
  - všesměrové vysílání (broadcast)
    - zprávu posíláme “**všem**” procesům
    - tj. **více** nespecifikovaným příjemcům
  
  - pozn.: další varianta - anycast (IPv6)
-

---



# Poznámky

Většina systémů umožňuje:

- odeslání zprávy skupině procesů
  - příjem zprávy od kteréhokoliv procesu
-



# Další otázky

- **vlastnosti fronty zpráv**
    - kolik jich může obsahovat, je omezená?
  - pokus **odeslat zprávu a fronta zpráv plná?**
    - většinou odesílatel pozastaven
  - **v jakém pořadí jsou zprávy doručeny?**
    - většinou v pořadí FIFO
  - jaké je **zpoždění** mezi odesláním zprávy a možností zprávu přijmout?
  - jaké mohou v systému nastat **chyby**, např. mohou se zprávy **ztrácet**?
-



# Délka fronty zpráv (buffering)

- **nulová délka**

žádná zpráva nemůže čekat  
odesílatel se zablokuje – “rendezvous”

- **omezená kapacita**

blokování při dosažení kapacity

- **neomezená kapacita**

odesílatel se nikdy nezablokuje

---



---

# Poznámka

- **Volbu** konkrétního chování primitiv send a receive provádějí **návrháři operačního systému**
  - Některé systémy nabízejí několik alternativních primitiv send a receive s různým chováním
-



---

# Terminologická poznámka

## neblokující send

- v některých systémech send, který se **vrací ihned** – ještě **před odesláním** zprávy
  - odeslání se provádí paralelně s další činností procesu
  - používá se zřídka
-





---

## Předpoklady pro další text

- send je neblokující, receive blokující
  - receive – umožňuje příjem od libovolného adresáta – `receive(ANY,zpráva)`
  - fronta zpráv – dostatečně velká na všechny potřebné zprávy
  - zprávy doručeny v pořadí FIFO a neztrácejí se
-



# Producent – konzument pomocí zpráv

- symetrický problém
- producent **generuje plné** položky
  - pro využití konzumentem
- konzument **generuje prázdné** položky
  - pro využití producentem

Úlohu producent/konzument jsme řešili s využitím semaforů,  
monitorů, nyní tedy i zasíláním zpráv



```
cobegin
```

```
while true do { producent }  
begin
```

```
    produkuje záznam;
```

```
    receive(konzument, m);
```

```
        // čeká na prázdnou položku
```

```
    m := záznam;
```

```
        // vytvoří zprávu
```

```
    send(konzument, m);
```

```
        // pošle položku konzumentovi
```

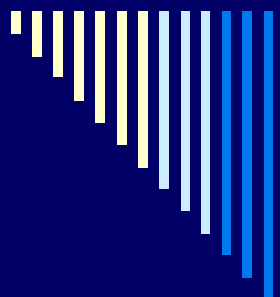
```
end {while}
```

```
||
```

Blokující  
operace

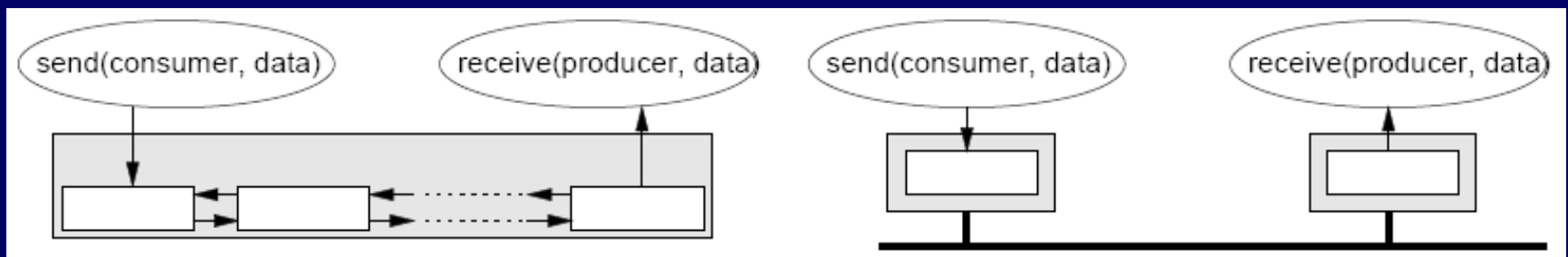
```
for i:=1 to N do                                { inicializace }
    send(producent, e);
    // pošleme N prázdných položek
while true do                                    { konzument }
begin
    receive(producent, m);
    // přijme zprávu obsahující data
    záznam := m;
    send(producent, e);
    // prázdnou položku pošleme zpět
    zpracuj záznam;
end {while}
coend.
```

Blokující  
operace



# Komunikující procesy

procesy nemusejí být na stejném stroji, ale mohou komunikovat po síti





# Problém určení adresáta

- dosud – zprávy posíláme procesům
- jak určit adresáta, pojmenovat procesy
  
- procesy nejsou **trvalé entity**
  - v systému vznikají a zanikají
- nebo více instancí **stejného** programu
  
- řešení – adresujeme **frontu zpráv**
  - nepřímá komunikace

Chtěli bychom např. poslat zprávu webovému serveru *Apache*, ale při každém spuštění stroje bude mít jiný PID

Neadresujeme proces, ale frontu zpráv



---

# Adresování fronty zpráv

- proces pošle zprávu
    - zpráva se připojí k určené frontě zpráv  
(vložení zprávy do fronty)
  
  - jiný proces přijme zprávu
    - vyjme zprávu z dané fronty
-



# Mailbox, port

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

## □ mailbox

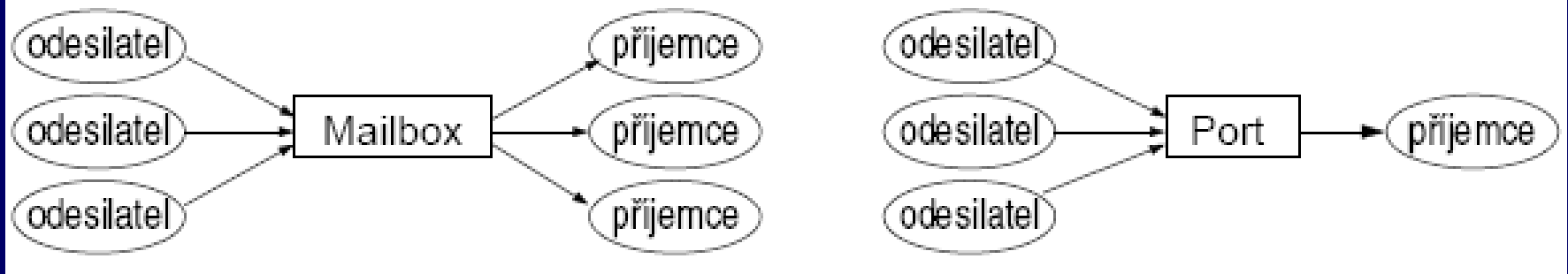
- fronta zpráv využívaná **více** odesílateli a příjemci
- obecné schéma
- operace receive – drahá, zvláště pokud procesy běží na různých strojích

## □ port

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



# Mailbox, port





# Implementace mechanismu zpráv – další problémy

- problémy, které nejsou u semaforů ani monitorů, zvláště při komunikaci po síti
  - **ztráta zprávy**
    - **potvrzení** o přijetí (acknowledgement)
    - pokud vysílač nedostane potvrzení do nějakého časového okamžiku (**timeout**), zprávu pošle znovu
  - **ztráta potvrzení**
    - zpráva dojde ok, ztratí se potvrzení
    - **číslování zpráv**, duplicitní zprávy se ignorují
-



# Problém autentizace

## □ problém autentizace

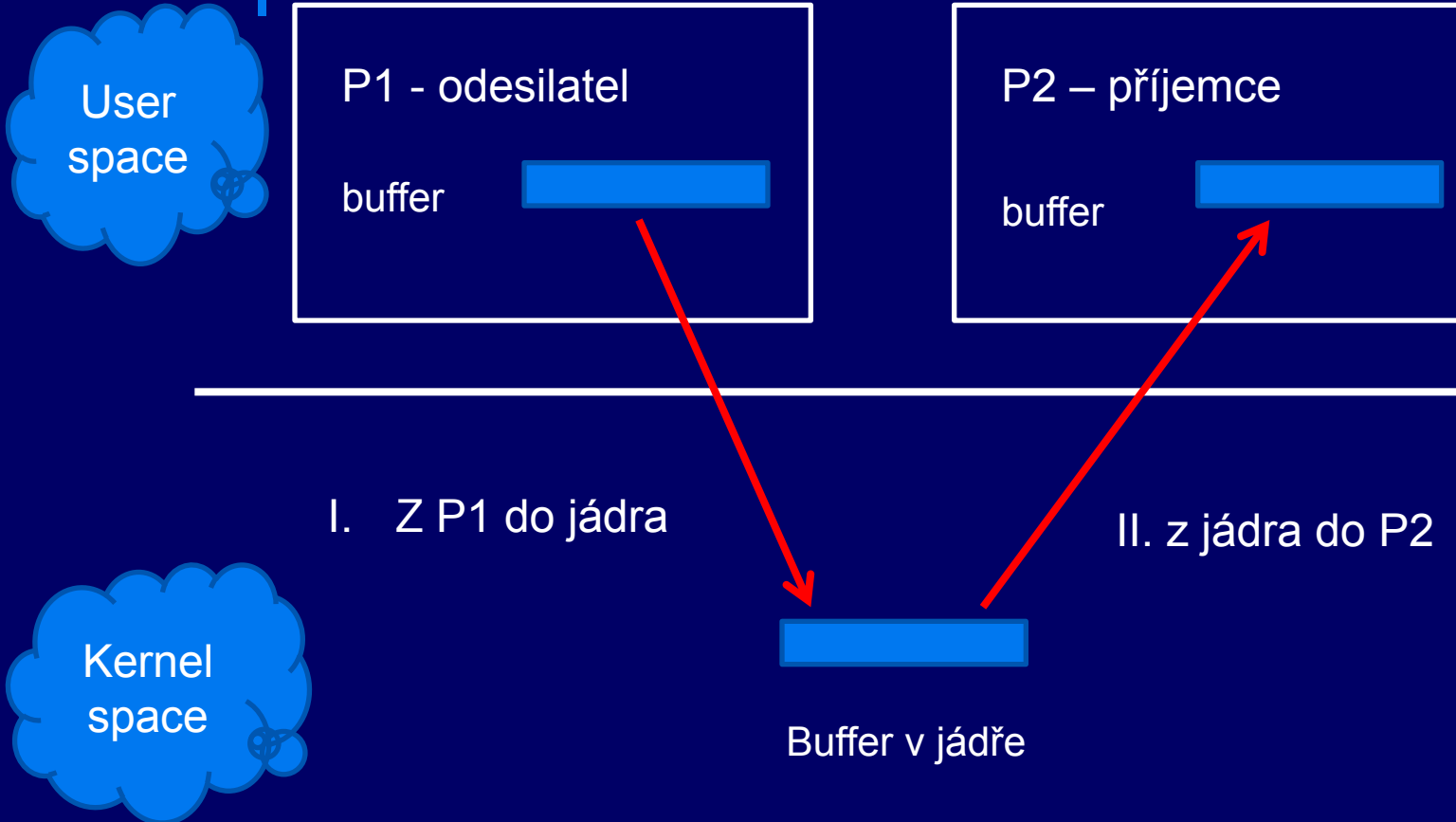
- ověřit, že nekomunikují s podvodníkem
- zprávy je možné **šifrovat**
- **klíč** známý pouze autorizovaným uživatelům (procesům)
- zašifrovaná zpráva obsahuje **redundanci**
  - umožní detekovat změnu zašifrované zprávy
- Pozn. Symetrické a asymetrické šifrování, podpisy zpráv



# Lokální komunikace (!)

- Na stejném stroji – snížení režie na zprávy
  - **Dvojití kopírování (!)**
    - z procesu odesílatele do fronty v jádře
    - z jádra do procesu příjemce
  - **rendezvous**
    - **eliminuje frontu zpráv**
    - send zavolán dříve než receive – odesílatel zablokován
    - vyvolán send i receive – zprávu zkopírovat z odesílatele **přímo** do příjemce
    - efektivnější, ale méně obecné (např. jazyk ADA)
-

# Dvojí kopírování (!!!)





# Lokální komunikace II.

- využití mechanismu **virtuální paměti (!)**
  - paměť obsahující zprávu je **přemapována**
    - z procesu odesílatele
    - do procesu příjemce
  - zpráva se **nekopíruje**
    - Virtuální paměť umí “čarovat“ komu daný kus fyzické paměti namapuje a na jaké adresy



# Typické aplikace

- struktura klient – server
    - klient požaduje vykonat práci
    - server práci vykonává
  - klienti vyžadují od serveru data nebo provedení operace nad daty
  
  - WWW klient x WWW server (Apache, IIS)
  - účetní aplikace x databázový systém (Oracle, MySQL)
  - klientský OS x souborový server (AFS)
-

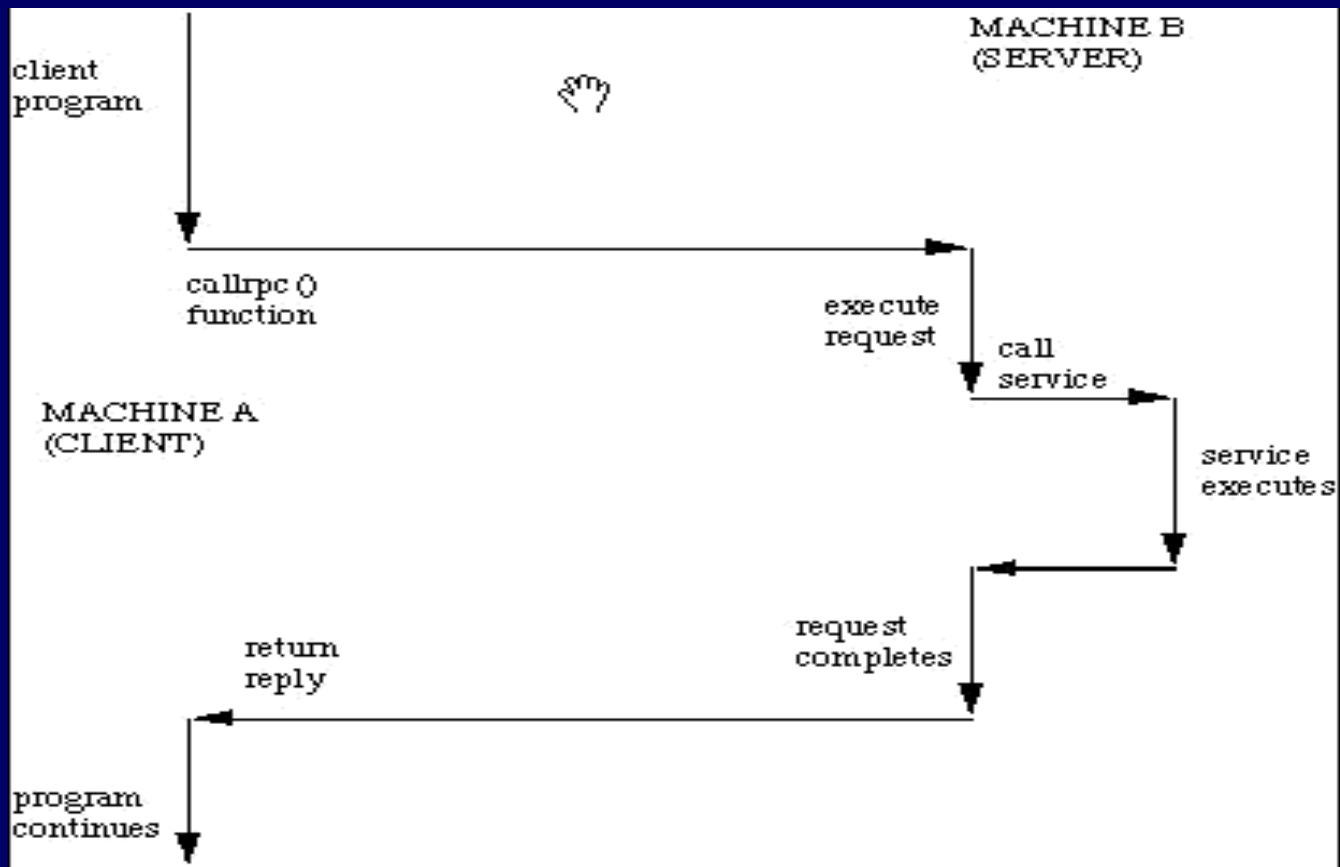


# Volání vzdálených procedur (RPC)

- používání send receive – opět nestrukturované
  - Birell a Nelson (1984)
  - dovolit procesům (klientům) volat procedury umístěné v jiném procesu (serveru)
  - mechanismus RPC (Remote Procedure Call)
  - variantou RPC je i volání vzdálených metod (Remote Method Invocation, RMI) v OOP, např. Javě
  - snaha aby co nejvíce připomínalo lokální volání
-



# RPC





# spojka klienta, serveru

- klientský program sestaven s knihovní funkcí, tzv. **spojka klienta (client stub)**
    - reprezentuje vzdálenou proceduru v adresním prostoru klienta
    - stejné jméno, počet a typ argumentů jako vzdálená procedura
  - program serveru sestaven se **spojkou serveru (server stub)**
  - spojky zakrývají, že volání není lokální
-

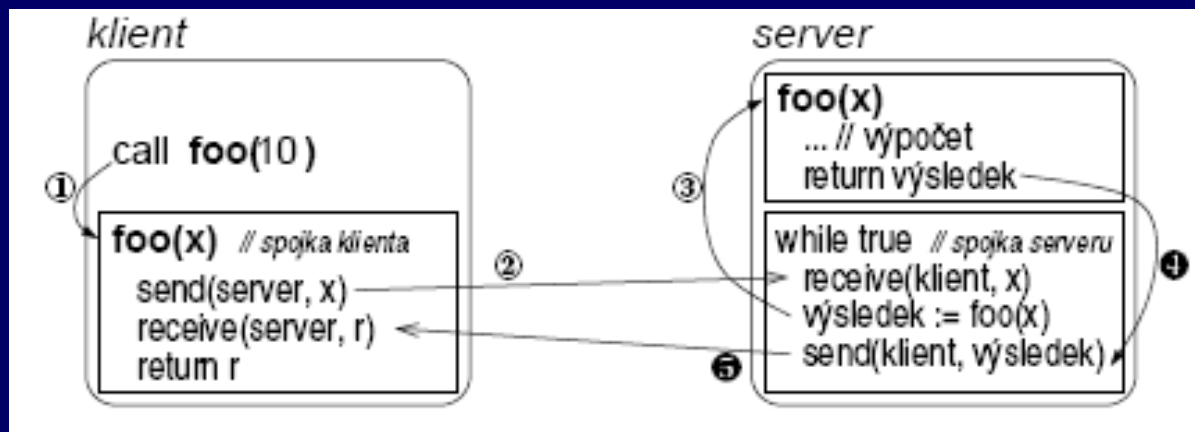
# Kroky komunikace



1. Klient zavolá **spojku klienta**, reprezentující vzdálenou proceduru
2. **Spojková procedura** argumenty zabalí do **zprávy**, pošle ji serveru
3. **Spojka serveru** zprávu přijme, vezme argumenty a zavolá **proceduru**
4. **Procedura** se vrátí, návrat. hodnotu pošle **spojka serveru** zpět klientovi
5. **Spojka klienta** přijme **zprávu** obsahující **návrat. hodnotu** a předá ji volajícímu

# Příklad

- volání `foo(x: integer): integer`



1. Klient volá spojku klienta `foo(x)` s argumentem `x=10`.
2. Spojka klienta vytvoří zprávu a pošle jí serveru:  
`procedure foo(x: integer):integer;`  
`begin`  
`send(server, m);` // zpráva obsahuje argument, tj. hodnotu "10"
3. Server přijme zprávu a volá vzdálenou proceduru:  
`receive(klient, x);` // spojka přijme zprávu, tj. hodnotu "10"  
`vysledek = foo(x);` // spojka volá fci `foo(10)`
4. Procedura `foo(x)` provede výpočet a vrátí výsledek.
5. Spojka serveru výsledek zabalí do zprávy a pošle zpět spojce klienta:  
`send(klient, vysledek);`
6. Spojka klienta výsledek přijme, vrátí ho volajícímu (jako kdyby ho spočetla sama):  
`receive(server, vysledek);`  
`foo = vysledek;`  
`return;`



# RPC – více procedur

- rozlišeny číslem
- spojka klienta ve zprávě předá kromě parametrů i číslo požadované procedury

Na serveru:

```
while true do begin
```

```
  receive(klient, m); // zpráva obsahující č. procedury a parametry
```

```
  if (m.číslo_procedury == 1) then výsledek = foo(m.x);
```

```
  if (m. číslo_procedury == 2) then výsledek = bar(m.x);
```

```
  ...
```

```
  send(klient, výsledek); // odešli zpět návratovou hodnotu
```

```
end
```

---



# RPC

dnes nepoužívanější jazyková konstrukce pro implementaci distribuovaných systémů a programů bez explicitního předávání zpráv

- DCE RPC, Java RMI, CORBA



# Programování RPC

- **Jazyk IDL** (Interface Definition Language)
    - Definujeme rozhraní mezi klientem a serverem (datové typy, procedury)
  - **Kompilátor** jazyka IDL
    - Vygeneruje spojky pro klienta i server
  - Server sestavíme se **spojkou serveru**
  - **Spojka klienta**
    - Podoba knihovny
    - Sestavujeme s ní klientské programy
-





# Problémy RPC

Volání vzdálené procedury přináší problémy, které při lokálním přístupu nejsou

## □ Parametry předávané odkazem

- Klient a server – různé adresní prostory
- Odeslání ukazatele nemá smysl
- Pro jednoduchý datový typ, záznam, pole – trik
  - Spojka klienta pošle odkazovaná data spojce serveru
  - Spojka serveru vytvoří nový odkaz na data atd.
  - Modifikovaná data pošle zpátky na klienta
  - Spojka klienta přepíše původní data

## □ Globální proměnné

- Použití není možné x lokálních procedur



# Reprezentace informace

- Společný problém pro předávání zpráv i RPC
- Stroje **různé** architektury
  - Může se lišit vnitřní reprezentace datových typů
  - **Kódování řetězců**
    - Udaná délka nebo ukončovací znak
    - Kódování jednotlivých znaků
  - **Numerické typy**
    - Způsob uložení (little endian, big endian)
    - Velikost (integer 32 nebo 64 bitů)

Problém  
může  
představovat  
komunikace  
mezi  
heterogenní  
mi systémy



# Little & big endian

- Chceme uložit: 4a3b2c1d (32bit integer)
  - **Big endian**
    - Nejvýznamnější byte (**MSB**) na nejnižší adrese
    - Motorola 68000, SPARC, System/370
    - V paměti od nejnižší adresy: 4a, 3b, 2c, 1d
  - **Little endian**
    - Nejméně významný byte (**LSB**) na nejnižší adrese
    - **Intel x86**, DEC VAX
    - V paměti od nejnižší adresy: 1d, 2c, 3b, 4a
-



# Další varianty endians

## □ Bi-endian

- Lze nastavit (např. mode bit), jaký formát uložení se bude používat
- Např. IA-64, defaultně little-endian

## □ Middle-endian

- Starší formát, jen poznámka
  - Např. 3b, 4a, 1d, 2c
-



# Otestování sw (jen ukázka)

```
#define LITTLE_ENDIAN 0
#define BIG_ENDIAN 1

int machineEndianness() {
    short s = 0x0102;
    char *p = (char *) &s;
    if (p[0] == 0x02)
        return LITTLE_ENDIAN;
    else
        return BIG_ENDIAN; }
```



# Řešení portability

- Definovat, jak budou data reprezentována při přenosu mezi počítači – **síťový formát**
    - Před odesláním do síťového formátu
    - Po přijetí do lokálního formátu
  - **Problém rozdílné velikosti**
    - Nová množina numerických typů, stejná velikost na všech podporovaných architekturách
    - **Int32\_t** – integer 32bitů, <stdint.h>, ISO C99
-



# Síťový formát (jen pro doplnění)

## □ TCP/IP

- **Network byte order** (big endian)
- Celé číslo – nejvýznamější byte jako první (**MSB**)

□ **Konverzní funkce** např. `<netinet/in.h>`

□ **htonl, htons**

□ **ntohl, ntohs**

□ (“host to net, net to host, short/long)

---



# Sémantika volání RPC

- **lokální volání** fce – právě jednou
  
  - **vzdálené volání**
    - chyba při síťovém přenosu (tam, zpět)
    - chyba při zpracování požadavku na serveru
    - klient neví, která z těchto chyb nastala
    - volající havaruje po odeslání zprávy před získáním výsledku
-





# Sémantika volání RPC

- právě jednou
  - alespoň 1x
    - opakované volání po timeoutu
    - dle charakteru operace
  - nejvýše 1x
    - klient volání neopakuje
    - při timeoutu – chyba, ošetření výjimek
-



# Idempotentní operace

- operace, kterou lze **opakovat** se stejným efektem, jaký mělo její první provedení
- pro sémantiku alespoň 1x
- $x = x + 10$  vs.  $x = 20$
- vypinac (zapni), vypinac (vypni) x vypinac (prepni)



# Ekvivalenty uvedených primitiv

- Lze implementovat semaforey pomocí zpráv a zprávy pomocí semaforů, ..., ?
- Tj. má obojí stejnou vyjadřovací sílu?

## Zprávy pomocí semaforů

- Využijeme řešení problému producent-konzument
  - **send:** Vložení zprávy do bufferu
  - **receive:** Vyjmutí zprávy z bufferu
-



# Semafor pomocí zpráv

## Semaforey pomocí zpráv

- Pomocný synchronizační proces (SynchP)
    - Pro každý semafor udržuje **čítač** (hodnotu semaforu)
    - A **seznam blokováných** procesů
  - Operace P a V
    - Jako funkce, které provedou **odeslání** požadavku
    - Poté čekají na odpověď pomocí **receive**
  - SynchP – v jednom čase jeden požadavek
    - Tím zajištěno vzájemné vyloučení
-



# Semafor pomocí zpráv

- Pokud SynP obdrží požadavek na operaci P
  - a čítač semafor  $> 0$ , odpoví ihned
  - Jinak neodpoví – čímž volajícího zablokuje
  
- Pokud SyncP obdrží požadavek na operaci V
  - A je blokový proces
    - Jednomu blokovánému odpoví, čímž ho vzbudí



# Stejná vyjadřovací síla

- Lze ukázat, že je možné implementovat
    - Semaforey pomocí monitoru
    - Monitory pomocí semaforů
  
  - Všechna dříve uvedená primitiva mají **stejnou vyjadřovací sílu**
  
  - Platí to i o mutexech? Ano, někdy..
-



# Semafor pomocí mutexů

□ Např. Barz (1983)

```
type semaphore = record
```

```
    val: integer;
```

```
    m: mutex;           // pro vzájemné vyloučení
```

```
    d: mutex;           // pro blokování (delay)
```

```
end;
```

```
procedure Initsem(var s: semaphore, count: integer);
```

```
begin
```

```
    s.val:=count;
```

```
    s.m :=ODEM;           // odemčeno
```

```
if count=0 then s.d := ZAM // zamčeno, někdo musí provést V(s)
```

```
    else s.d := ODEM     // odemčeno
```

```
end;
```

---



---

```
procedure P(var s: semaphore);
```

```
begin
```

```
    mutex_lock(s.d); // když s.val=0, čekáme
```

```
    mutex_lock(s.m); // kritická sekce –
```

```
        // přístup k s.val
```

```
s.val := s.val - 1;    // vždy bude platit s.val >= 0
```

```
if s.val > 0 then
```

```
    mutex_unlock(s.d); // další proces do operace P
```

```
mutex_unlock(s.m)    // konec přístupu k val
```

```
end;
```

---





---

```
procedure V(var s: semaphore);
```

```
begin
```

```
    mutex_lock(s.m);
```

```
    s.val := s.val + 1;
```

```
    if s.val = 1 then
```

```
        mutex_unlock(s.d)
```

```
    mutex_unlock(s.m)
```

```
end;
```

---



## Omezení mutexů v některých implementacích

Z důvodů efektivity někdy omezení mutexů:

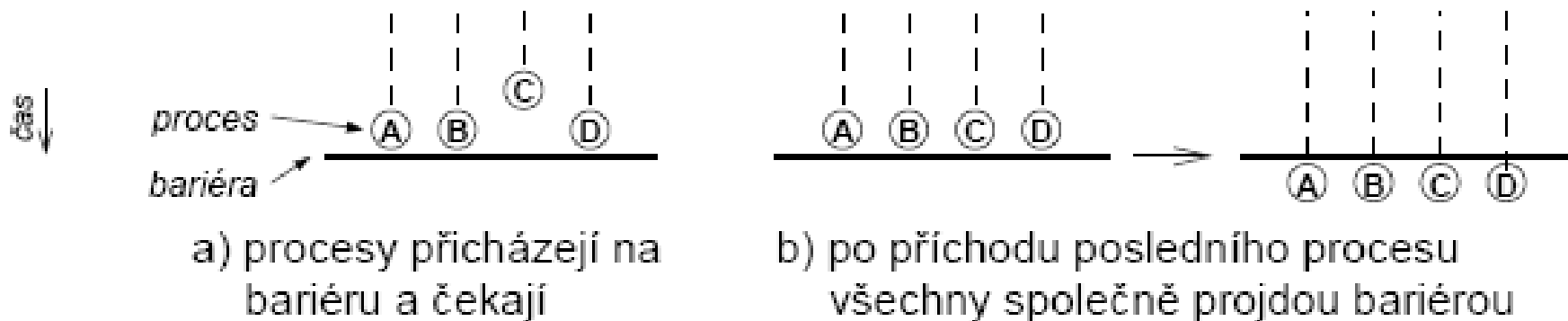
- Mutex smí odemknout pouze vlákno, které předtím provedlo jeho uzamknutí (POSIX.1)
  - Nelze použít pro implementaci obecných semaforů
  - Potom slabší než výše uvedená primitiva



# Bariéry

- Synchronizační mechanismus pro **skupiny procesů**
  - Použití ve vědecko-technických výpočtech
  - Aplikace – skládá se **z fází**
    - Žádný proces nesmí do následující fáze dokud všechny procesy nedokončily fázi předchozí
  - Na konci každé fáze – synchronizace na **bariéře**
    - Volajícího **pozastaví**
    - Dokud všechny procesy také nezavolají barrier
  - Všechny procesy **opustí bariéru současně**
-

# Bariéra



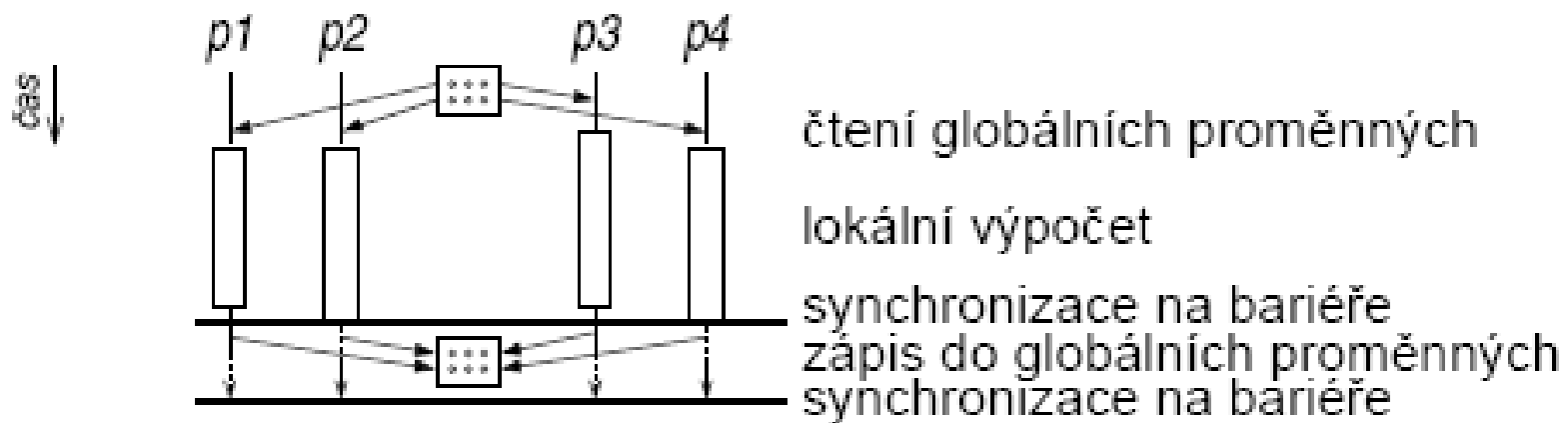


---

# Bariéra – iterační výpočty

- Jednotlivé kroky výpočtu
  - Matice  $X(i+1)$  z matice  $X(i)$
  - Každý proces počítá 1 prvek nové matice
  - Synchronizace pomocí bariery
-

# Bariera – iterační výpočty





# Klasické problémy IPC

IPC – Interprocess Communication

- Producent- konzument
- Večeřící filozofové
- Čtenáři - písaři
- Spící holič
- Řada dalších úloh

Vždy když někdo vymyslí nový synchronizační mechanismus, otestuje se jeho použitelnost na těchto klasických úlohách



# Problém večeřících filozofů

- Dijkstra 1965, dining philosophers
- Model procesů **soupeřících o výhradní přístup** k omezenému počtu zdrojů
  - Může dojít k zablokování, vyhladovění
- „Test elegance“ nových synchronizačních primitiv

Zablokování a vyhladovění jsou dva rozdílné pojmy,  
uvidíme dále..





---

# Problém večeřících filozofů

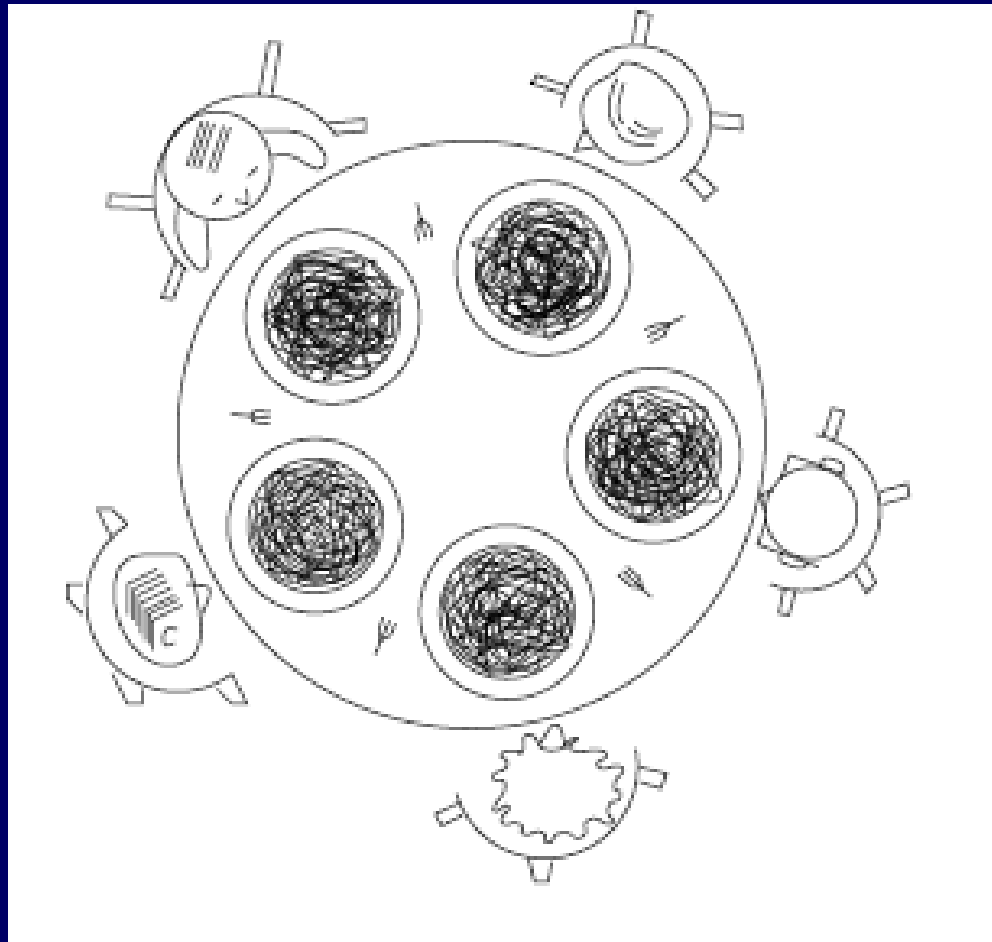
- 5 filozofů sedí kolem kulatého stolu
- Každý filozof má před sebou talíř se špagetami
- Mezi každými dvěma talíři je vidlička
- Filozof potřebuje dvě vidličky, aby mohl jíst

Ukázka:

<http://webplaza.pt.lu/~onarat/>

---

# Problém večerících filozofů





# Problém večeřících filozofů

- Život filozofa – jí a přemýšlí
    - Tyto fáze se u každého z nich střídají
  - Když dostane hlad
    - Pokusí se vzít si dvě vidličky
      - Uspěje – nějakou dobu jí, pak položí vidličky a pokračuje v přemýšlení
  - Úkolem
    - Napsat program pro každého filozofa, aby pracoval dle předpokladů a nedošlo k potížím
    - aby se každý najedl
-



---

# Problém večeřících filozofů

Očísujeme **filozofy**: 0 .. 4

Očísujeme **vidličky**

filozof 0: levá vidlička 0, pravá 1

filozof 1: levá vidlička 1, pravá 2

...

Procedura **zvedni(v)**

- Počká, až bude vidlička k dispozici a pak jí zvedne
-



# Kód filozofa - chybný

```
const N = 5;  
procedure filosof(i: integer);  
begin  
    premyslej();  
    zvedni (i);  
    zvedni ((i+1) mod N);  
    jez();  
    poloz(i);  
    poloz((i+1) mod N)  
end;
```

Všichni filozofové  
běží dle stejného  
kódu, např. 5 vláken,  
každé vykonává kód:

```
filosof(0)  
filosof(1)  
...  
filosof(4)
```



# Problém uvíznutí (deadlock)

Popis chyby:

Všichni filozofové zvednou najednou levou vidličku, žádný z nich už nemůže pokračovat, dojde k deadlocku

**Deadlock:**

cyklické čekání dvou či více procesů na událost, kterou může vyvolat pouze některý z nich, nikdy k tomu však nedojde

$f[0]$  čeká, až  $f[1]$  položí vidličku,  $f[1]$  čeká na  $f[2]$ ,  
2-3, 3-4, 4-0 = cyklus

---

---



# Modifikace algoritmu

Filozof zvedne levou vidličku a zjistí, zda je pravá vidlička dostupná.

Pokud není, položí levou, počká nějakou dobu a zkusí znovu.

---



# Stále chybná ...

Pokud by filozofové vzali najednou levou vidličku, budou běžet cyklicky (vidí, že pravá není volná, položí..)

Proces není blokováný (x od deadlock), ale běží bez toho, že by vykonával užitečnou činnost

Analogie – situace „až po vás“ přednost ve dveřích

**Vyhladovění** (starvation)

proces se nedostane k požadovaným zdrojům

---





# Řešení pomocí monitoru

Když chce *i*-tý filozof jíst,  
zavolá funkci `chci_jist(i)`

Když je najezen, zavolá `mam_dost(i)`

Ochrana před uvíznutím:

obě vidličky musí zvednout najednou, v kritické sekci,  
uvnitř monitoru

Řešení je vícero, následující např. nezabrání dvěma  
konspirujícím filozofům, aby bránili jíst třetímu

---

```
Monitor vecerici_filozofove;
const N=5;
var
    f : array [0 .. N-1] of integer; {počet vidliček dostupných filozofovi}
    a : array [0 .. N-1] of integer; {podmínka vidličky k dispozici}
    i : integer;

procedure chci_jist ( i: integer)
begin
    if f [ i ] < 2 then a [ i ].wait;
    decrement ( f [ (i-1) mod N] ); { sniž o jedna vidličky levému }
    decrement ( f [ (i+1) mod N] ); { sniž o jedna vidličky pravému}
end;
```

```
procedure mam_dost ( i: integer)
```

```
begin
```

```
    increment( f [ (i-1) mod N] ); { zvětši o jedna vidličky levému }
```

```
    increment ( f [ (i+1) mod N] ); { zvětši o jedna vidličky pravému }
```

```
    if f [ (i-1) mod N] = 2 then { má levej soused 2 vidličky? }
```

```
        a [ (i-1) mod N] . signal;
```

```
    if f [ (i+1) mod N] = 2 then { má pravej soused 2 vidličky? }
```

```
        a [ (i+1) mod N] . signal;
```

```
end;
```

```
begin { inicializace monitoru, filozof má 2 vidličky }
```

```
    for i:=0 to 4 do
```

```
        f[i] := 2
```

```
end;
```