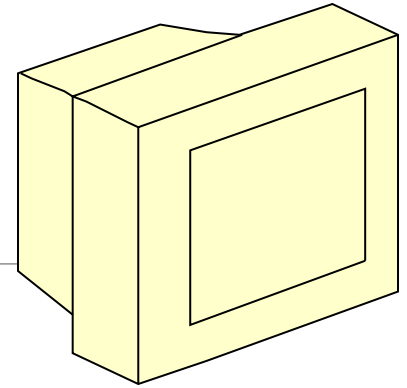


07. Meziprocesová komunikace

- zprávy, RPC

ZOS 2024, L. PEŠIČKA

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?

Opakování

Kde je uložený PID?
V PCB.

proces má PID, vlákno má TID

- proces může mít více vláken
- každé vlákno – program counter PC (konkrétně např. CS:EIP) – ukazatel na prováděnou instrukci („program counter“)

hierarchie procesů

- proces si udržuje info o rodiči PPID - getppid()

proces – jednotkou přidělování prostředků

vlákno – jednotkou plánování

Opakování

- **fork()** – vytvoří duplicitní kopii aktuálního procesu
 - Kdo dostane návratovou hodnotu nulu?
 - Kdo nenulovou hodnotu a co tato hodnota vyjadřuje?
- **exec()** – nahradí program v aktuálním procesu jiným programem (exec family - např. `execv`, `execl`)
- **wait()** – rodič může čekat na dokončení potomka

příkaz strace – jaká systémová volání jsou vykonávána

strace ls je.txt není.txt

bude nás zajímat program ls:

- vidíme volání `execve("/bin/ls", ...)`
- vidíme chybový výstup `write(2, ...)`
- vidím standardní výstup `write(1, ...)`

strace

ukázka částí výstupu:

- volání execve
- write na deskriptor 1- standardní výstup, 2-chybový výstup

```
eryx2> strace ls je.txt neni.txt
execve("/bin/ls", ["ls", "je.txt", "neni.txt"], [/* 33 vars */]) = 0
brk(0)                                = 0x80632dc
access("/etc/ld.so.nohwcap", F_OK)    = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x55b3e000
```

```
write(2, ": No such file or directory"... , 27: No such file or directory) = 27
write(2, "\n"... , 1
)
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 4), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x55b3d000
write(1, "je.txt\n"... , 7je.txt
```

Meziprocesová komunikace

IPC – Interprocess Communication (!)

Komunikace procesů:

- **Sdílená paměť**
(předpoklad: procesy na stejném uzlu)
- **Zasílání zpráv**
(na stejném uzlu i na různých uzlech)

Meziprocesová komunikace

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

Linux - signály

Signály představují jednu z forem meziprocesové komunikace

- signál – speciální **zpráva** zaslaná jádrem OS procesu
 - Např. uživatel stiskne na klávesnici CTRL+C
- iniciátorem signálu může být i proces – viz **man 2 kill**
- zpráva neobsahuje jinou informaci než číslo signálu
- jsou **asynchronní** - mohou přijít kdykoliv, ihned jej proces obslouží (přeruší provádění kódu a začne obsluhovat signál)
- Signál je specifikován svým číslem, často se používají symbolická jména (SIGTERM, SIGKILL, ...) – viz **man 7 signal**

Linux - signály

příkaz	popis
ps aux	Informace o procesech
kill -9 1234	Pošle signál č. 9 (KILL) procesu s PID číslem 1234
man kill	Nápověda k příkazu kill
man 2 kill	Nápověda k volání kill
man 7 signal	Nápověda k signálům
kill -l	Vypíše seznam signálů

<https://stackoverflow.com/questions/45993444/in-detail-what-happens-when-you-press-ctrl-c-in-a-terminal>

Linux - signály

man 7 signal

Události **generující** signály

- Stisk kláves (*CTRL+C* generuje *SIGINT*)
- Příkaz kill (1), systémové volání kill (2)
- Mohou je generovat uživatelské programy přes systémové volání

Reakce na signály (!!)

- Standardní zpracování
 - ukončí proces (**Term**),
 - ukončí a provede dump (**Core**)
 - zastaví (**Stop**), pustí zastavený (**Cont**)
- Vlastní zpracování naší funkcí
- Ignorování signálu (ale některé nelze, např. SIGKILL, SIGSTOP)

Signály a vlákna (man 7 signal)

The signal disposition is a **per-process attribute**: in a multithreaded application, the disposition of a particular signal is the same for all threads.

A child created via `fork(2)` inherits a copy of its parent's signal dispositions. During an `execve(2)`, the dispositions of handled signals are reset to the default; the dispositions of ignored signals are left unchanged.

Stack overflow:

Note: Signals are delivered to **any thread** that is **not explicitly blocking** its delivery. This does not change that. You still need to use `pthread_kill()` or similar mechanisms to direct the signal to a **specific thread**; signals that are raised or **sent to the process** (instead of a specific thread), will still be **handled by a random thread** (among those that do not block it).

Signály a vlákna (man 7 signal)

<code>raise(3)</code>	Sends a signal to the calling thread.
<code>kill(2)</code>	Sends a signal to a specified process, to all members of a specified process group, or to all processes on the system.
<code>killpg(2)</code>	Sends a signal to all of the members of a specified process group.
<code>pthread_kill(3)</code>	Sends a signal to a specified POSIX thread in the same process as the caller.
<code>tgkill(2)</code>	Sends a signal to a specified thread within a specific process. (This is the system call used to implement <code>pthread_kill(3)</code> .)
<code>sigqueue(3)</code>	Sends a real-time signal with accompanying data to a specified process.

```
#!/bin/bash
```

```
obsluha() {  
    echo "Koncim..."  
    exit 1  
}
```

```
# pri zachyceni signalu SIGINT se vykona funkce: 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

SIGTERM	Žádost o ukončení procesu
SIGSEGV	Porušení segmentace paměti
SIGABORT	Přerušení procesu
SIGHUP	Odříznutí od terminálu (nohup ignoruje)
SIGKILL	Bezpodmínečné zrušení procesu
SIQQUIT	Ukončení terminálové relace procesu
SIGINT	Přerušení terminálu (CTRL+C)
SIGILL	Neplatná instrukce
SIGCONT	Navrácení z pozastavení procesu
SIGSTOP	Pozastavení procesu (CTRL+Z)

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from <code>abort(3)</code>
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from <code>alarm(2)</code>
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at terminal
SIGTTIN	21,21,26	Stop	Terminal input for background process
SIGTTOU	22,22,27	Stop	Terminal output for background process

The signals **SIGKILL** and **SIGSTOP** cannot be caught, blocked, or ignored.

Datové roury

- jednosměrná komunikace mezi 2 procesy
- data zapisována do roury jedním procesem lze dalším hned číst
- data čtena přesně v tom pořadí, v jakém byla zapsána

```
cat /etc/passwd | grep josef | wc -l
```

Datové roury

systémové volání **pipe**:

```
int pipe (int pipefd[2])
```

pipefd[0] .. odsud čteme (in)

pipefd[1] .. sem zapisujeme (out)

jednosměrný komunikační kanál mezi příbuznými procesy

Příklad použití roury (!)

```
int main() { ...
```

```
int pipefd[2];
```

```
pipe(pipefd);
```

```
if (fork() == 0) {
```

```
    /* potomek z roury čte */
```

```
    close (pipefd[1]); // zavřeme výstup
```

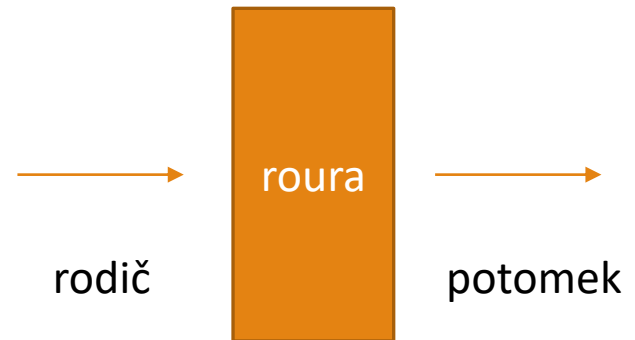
```
    ... read(pipefd[0], ...); ...           //čteme ze vstupu
```

```
} else {
```

```
    /* rodič do roury zapisuje */
```

```
    close (pipefd[0]);           // zavřeme vstup
```

```
    ... write(pipefd[1], ...); }      //zapisujeme na výstup
```



Problém sdílené paměti

Vyžaduje **umístění** objektu ve **sdílené paměti**

Někdy není **vhodné**

- **Bezpečnost** – **globální data** přístupná kterémukoliv procesu/vlákně (co když nebudou řídit přístup semaforem, mutexem?)

Někdy není **možné**

- Procesy běží na **různých strojích**, komunikují spolu po síti

Jiný způsob IPC – předávání zpráv

Předávání zpráv – send, receive

Zavedeme 2 primitiva:

- | | |
|--|-------------------|
| <code>send (adresat, zprava)</code> | - odeslání zprávy |
| <code>receive(odesilatel, zprava)</code> | - příjem zprávy |

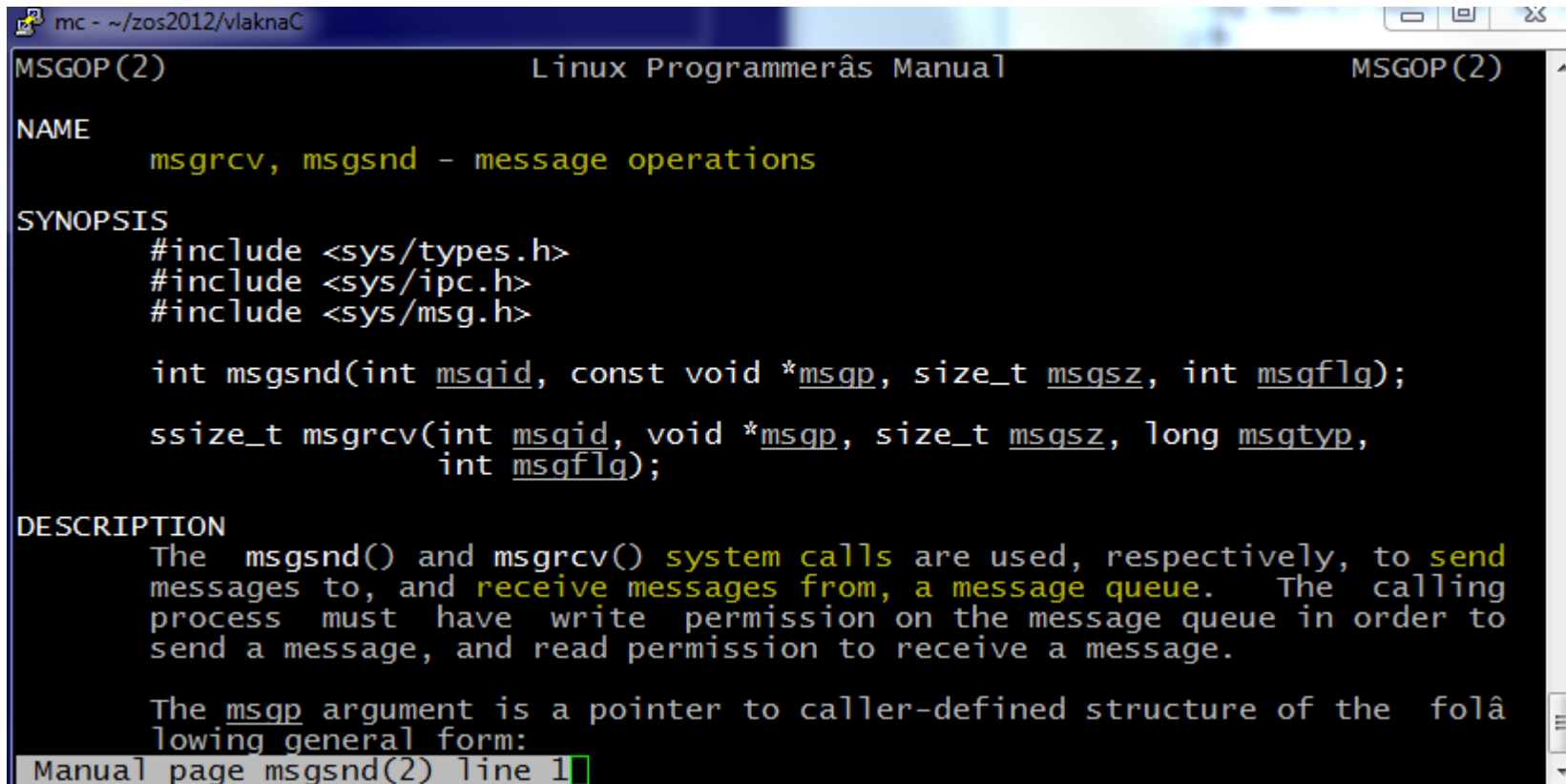
Send

- Zpráva (libovolný datový objekt) bude zaslána adresátovi

Receive

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

Linux – systémová volání



```
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
```

Linux – systémová volání

- `msgsnd()` – pošle zprávu do fronty zpráv
- `msgrcv()` – přijme zprávu z fronty zpráv

Send, receive – obecné názvy z teorie operačních systémů

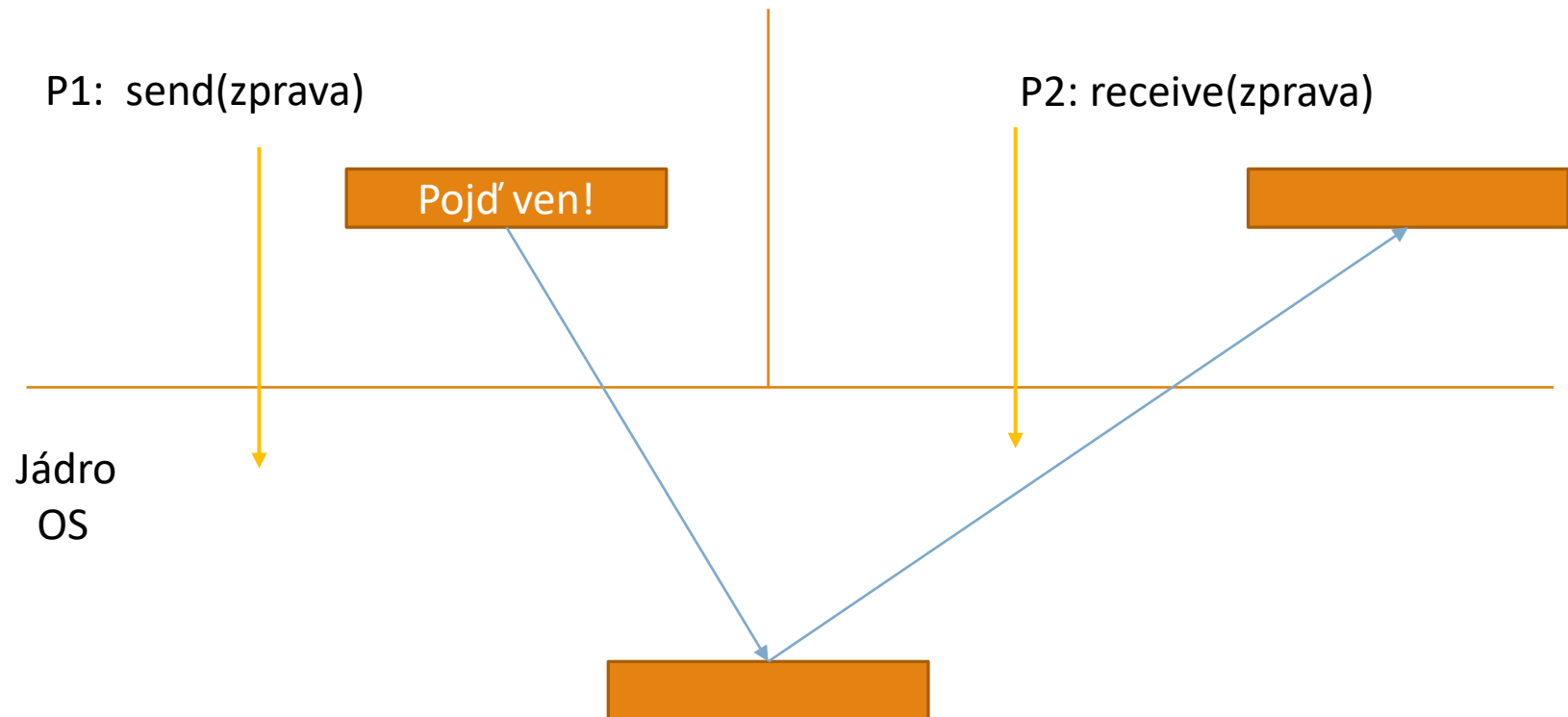
`msgsnd`, `msgrcv` – konkrétní implementace v případě Linuxu

Vlastnosti

- synchronizace (blokující nebo neblokující volání)
- unicast, multicast, broadcast
- přímá komunikace x nepřímá komunikace (adresujeme frontu)
- délka fronty zpráv (nulová, omezená, neomezená)
- pevná vs. proměnná délka zprávy

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

Volání jádra



Synchronizace

- blokující (**synchronní**)
 - Volání se zablokuje, dokud požadovaná událost nenastane
 - typicky **receive** čeká na zprávu
- neblokující (**asynchronní**)
 - Volání hned pokračuje dále
 - typicky **send** předá zprávu jádru a dál se nestará

Možné varianty send - receive

- blokující send
 - čeká na převzetí správy příjemcem
- **neblokující send** (nejčastěji)
 - vrací se ihned po odeslání zprávy
 - většina systémů
- **blokující receive** (nejčastěji)
 - není-li ve frontě žádná zpráva, zablokuje se
 - většina systémů
- neblokující receive
 - není-li zpráva, vrací chybu

Receive s omezeným čekáním

receive (odesílatel, zprava, t)

- čeká na příchod zprávy dobu t
- pokud zpráva nepřijde do doby t , vrací se volání s chybou

A horizontal orange scroll with a slight 3D effect, featuring a darker orange border and a small circular detail at the top right corner.

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

Většina systémů umožňuje:

- odeslání zprávy skupině procesů
- příjem zprávy od kteréhokoliv procesu

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

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 a čeká na příjemce – “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

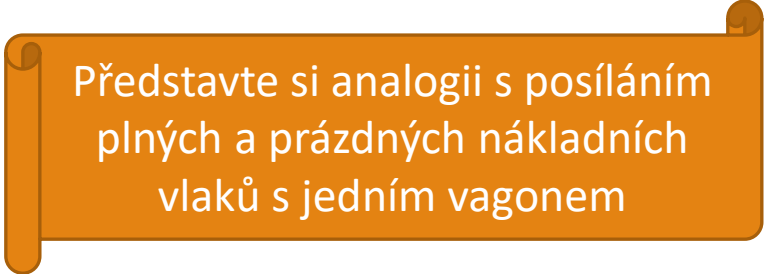
Některé systémy nabízejí několik alternativních primitiv send a receive s různým chováním

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



Představte si analogii s posíláním plných a prázdných nákladních vlaků s jedním vagonem

cobegin

while true do

{ producent }

begin

produkuj záznam;

Blokující
operace

receive(konzument, m);

// čeká na prázdnou položku

m = záznam;

// vytvoří zprávu

send(konzument, m);

// pošle položku konzumentovi

end {while}

||

```
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
```

```
    zaznam = m;
```

```
    send(producent, e);                        // prázdnou položku pošleme zpět
```

```
    zpracuj zaznam;
```

```
end {while}
```

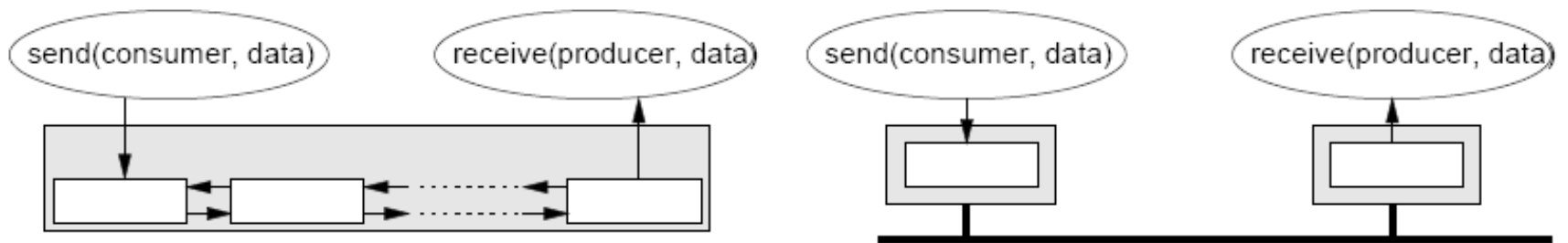
```
coend.
```

Blokující
operace



Komunikující procesy

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



Problém určení adresáta

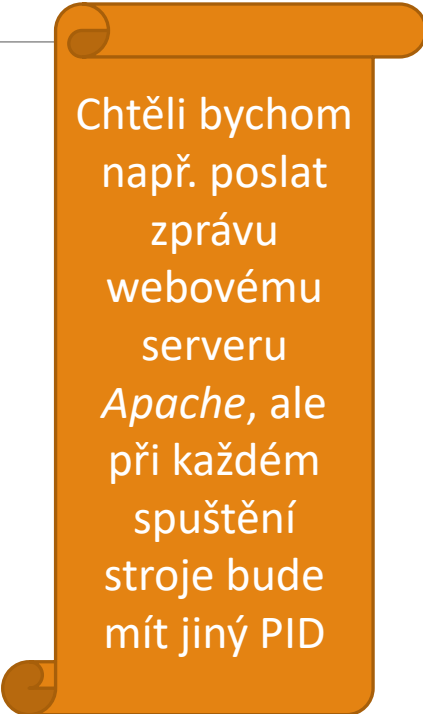
dosud – zprávy posíláme procesům
jak ale určit adresáta, pojmenovat procesy?

procesy nejsou **trvalé entity**


- v systému **vznikají a zanikají**

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

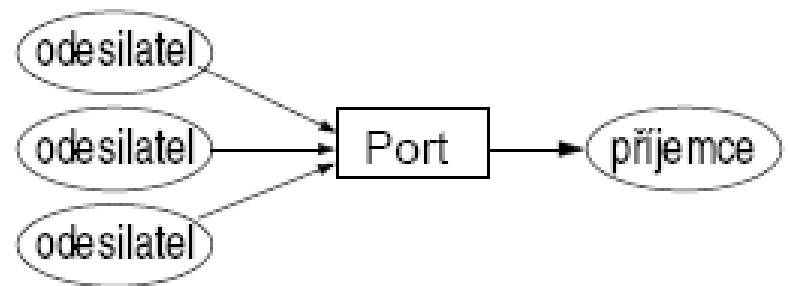
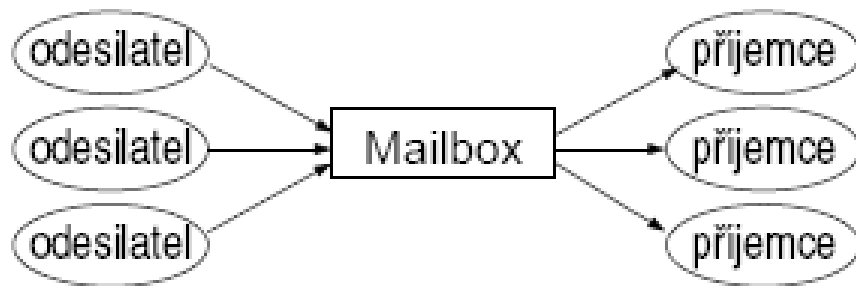
■ mailbox

- 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

Lokální komunikace (!)

Na stejném stroji – **snížení režie** na zprávy

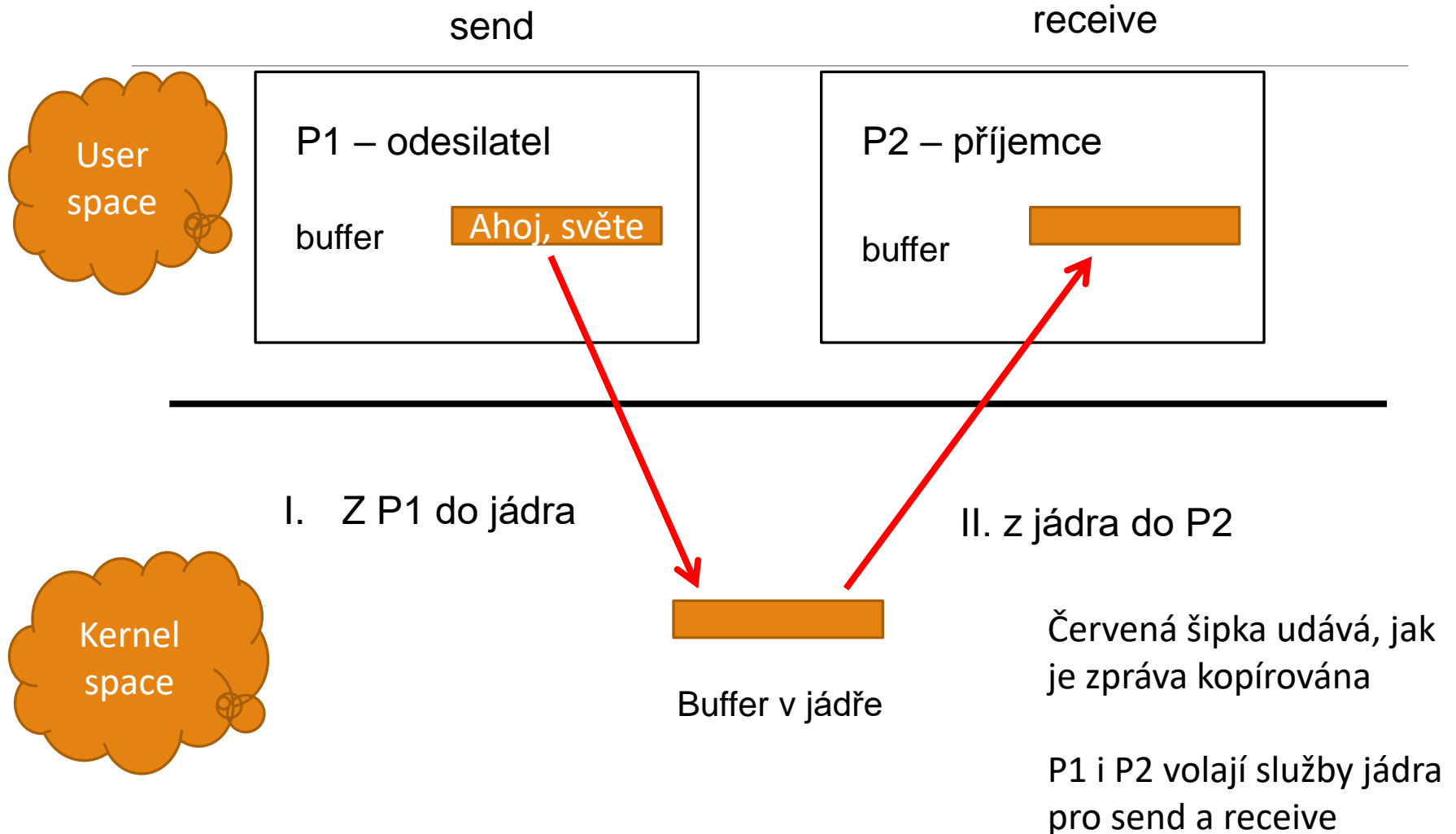
- **Dvojí kopírování (!)**

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

- **Rendezvous**

- **eliminuje frontu zpráv v jádře (1 kopírování)**
- např. send zavolán dříve než receive – odesílatel zablokován
- až jsou obě volání OS, send i receive – zprávu zkopírovat z odesílatele **přímo** do příjemce
- efektivnější, ale méně obecné (např. jazyk ADA)

Dvojí kopírování (!!!)



Lokální komunikace II.

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

využití mechanismu **virtuální paměti**

- lze využít např. při rendezvous
- paměť obsahující zprávu je **přemapována**
 - z procesu odesílatele
 - do procesu příjemce
- zpráva se **nekopíruje**
 - Virtuální paměť umí “čarovat” komu daný kus fyzické paměti namapuje a na jaké adresy
- často ale rendezvous místo přemapování používá jedno kopírování

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

local vs. remote call

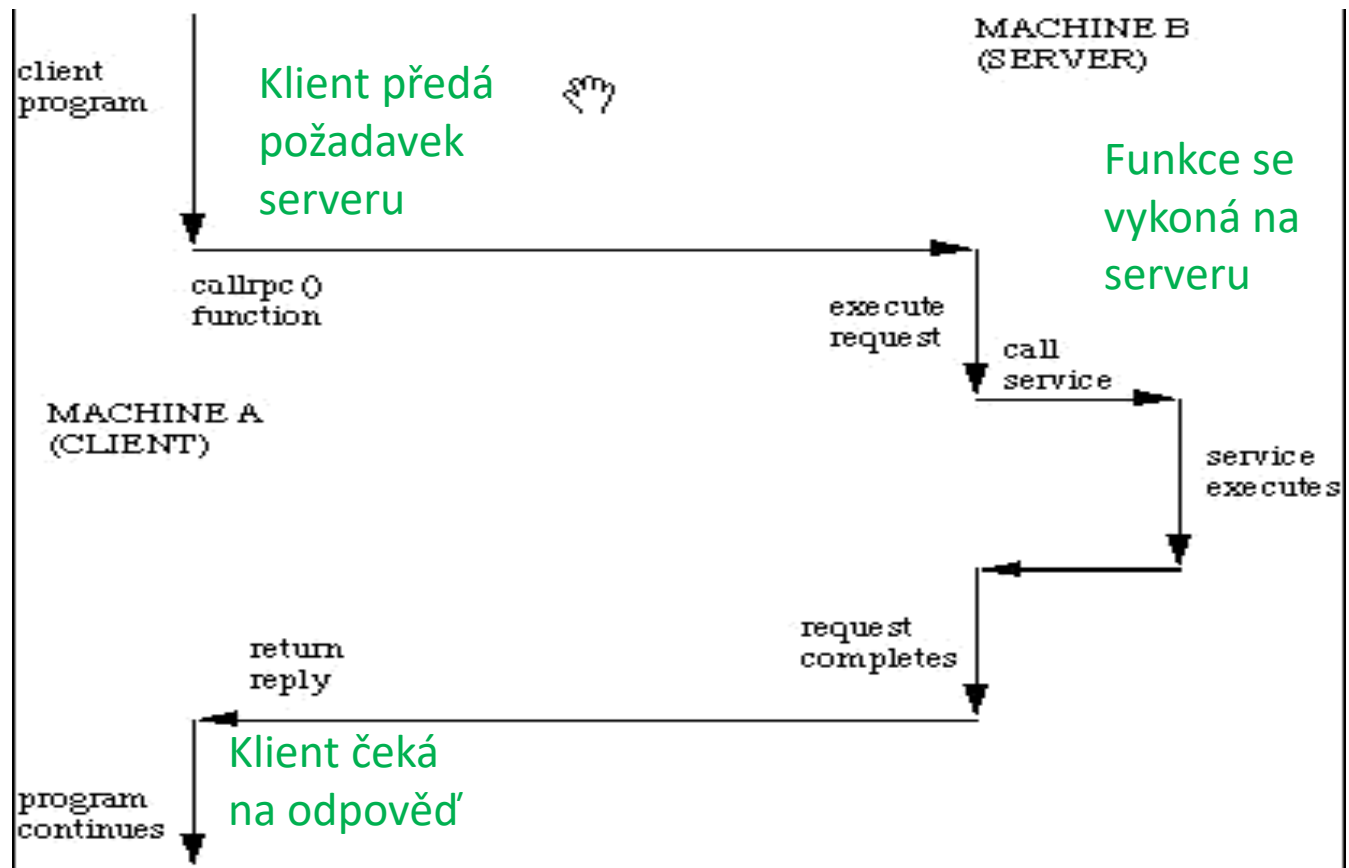
Local call:

```
int add (int a, int b) {  
    int result;  
  
    result = a + b;  
  
    return result;  
  
}
```

Remote (RPC) call:

```
int add (int a, int b) {  
    int result;  
  
    send ("server, call function add", a, b);  
  
    /* server provides the computation */  
  
    receive (result);  
  
    return result;  
  
}
```

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

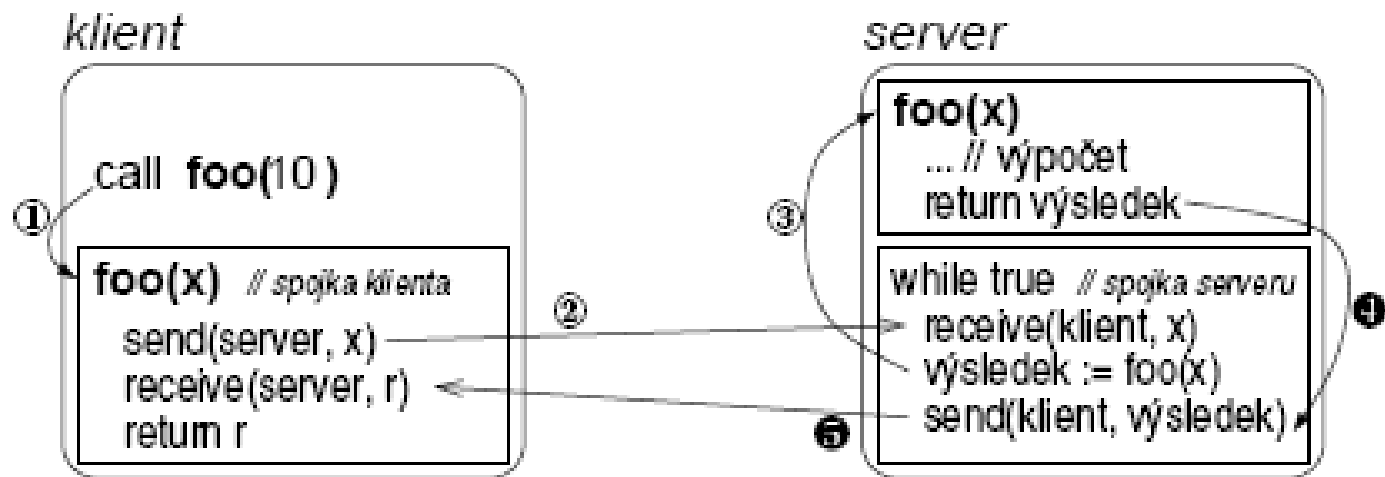
Kroky komunikace



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

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 vysledek = foo(m.x);
  if (m. číslo_procedury == 2) then vysledek = bar(m.x);
  ...
  send(klient, vysledek);      // odešli zpět návratovou hodnotu
end
```

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

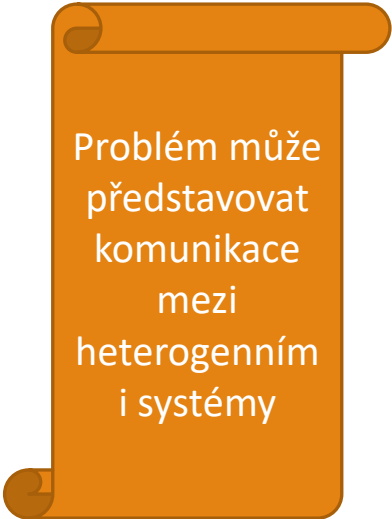
- Klient a server – **různé adresní prostory**
- Odeslání ukazatele nemá smysl
- Pro jednoduchý datový typ, záznam, pole – lze řešit
 - Spojka klienta **pošle odkazovaná** data spojce serveru
 - Spojka serveru vytvoří nový odkaz na data atd.
 - Modifikovaná data pošle zpátky na klienta
 - Spojka klienta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

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

Sémantika volání RPC

- **lokální volání** funkce – právě jednou
 - Lokálně voláme funkci `secti()`, ta se vykoná právě 1x
 - Bereme to jako samozřejmost
- **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
 - ideální stav
- 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

- **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 obslouží jeden požadavek**
 - Tím zajištěno vzájemné vyloučení

Semafor pomocí zpráv

Pokud SynP obdrží požadavek na operaci P

- a čítač pro daný semafor > 0 , odpoví ihned
- Jinak neodpoví – čímž volajícího zablokuje
(receive volajícího bude čekat na zprávu)

Pokud SyncP obdrží požadavek na operaci V

- A je blokový proces
- Jednomu blokovýmu odpoví, čímž ho vzbudí

Stejná vyjadřovací síla

Lze ukázat, že je možné implementovat

- Semaforey pomocí monitoru
- Monitory pomocí semaforů

Uvedená primitiva (semaforey, zprávy, monitory) mají **stejnou vyjadřovací sílu**

Platí to i o mutexech? Někdy..

Omezení mutexů

Z důvodů efektivity někdy omezení mutexů:

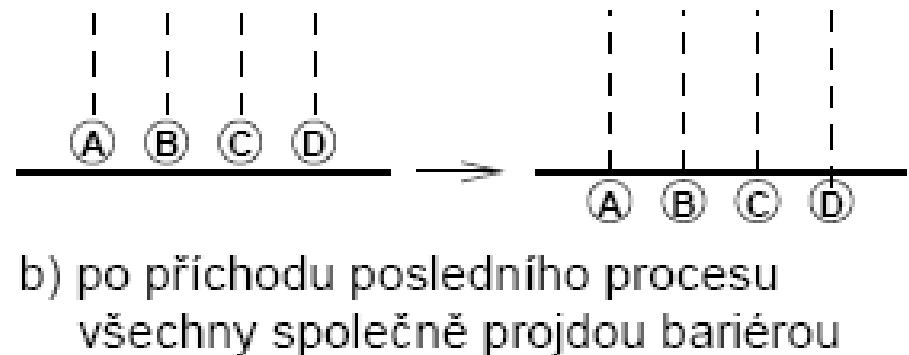
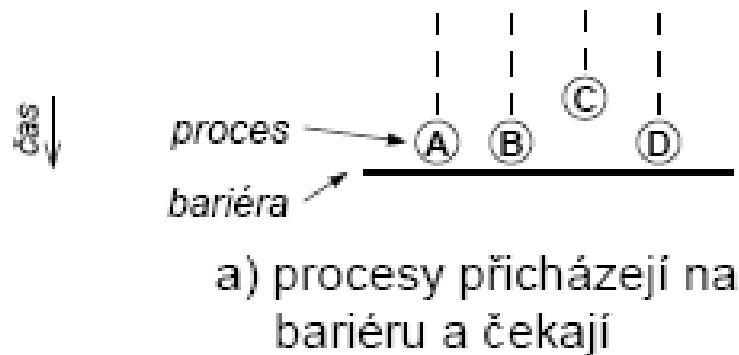
Mutex smí odemknout pouze vlákno, které předtím provedlo jeho uzamknutí (POSIX.1).

- Potom je slabší než ostatní uvedená primitiva

Bariéry

- Synchronizační mechanismus pro skupiny procesů
- 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

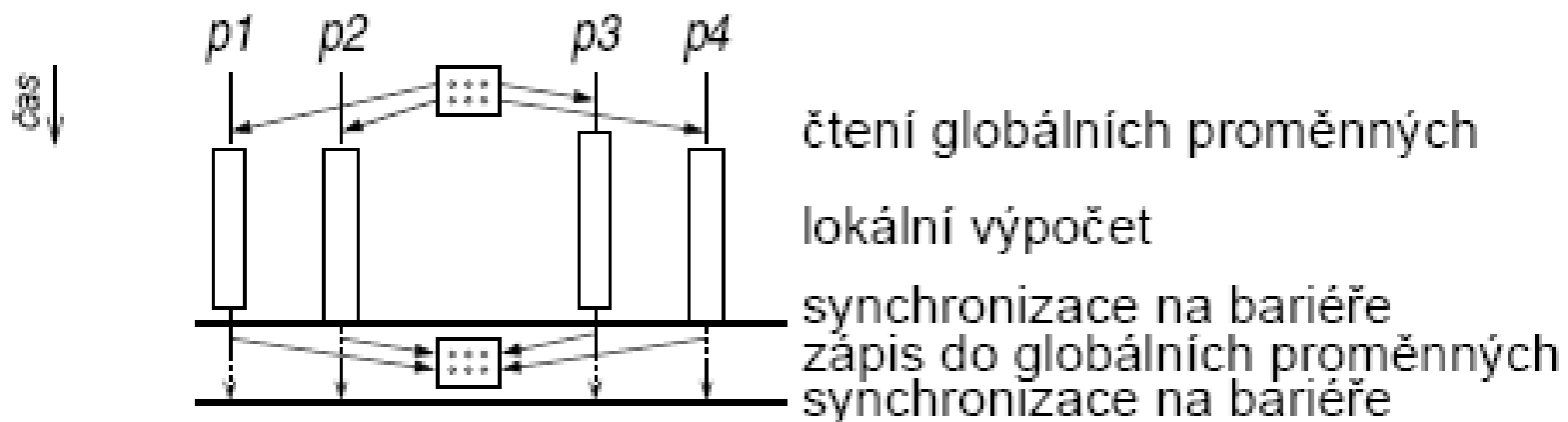
Bariéra



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



Bariéra pomocí monitoru v C

```
void bariera() {
    pthread_mutex_lock(&lock);
    barieraCount++;
    if (barieraCount != BAR_PRAH) {
        pthread_cond_wait(&cond, &lock);
    } else {
        pthread_cond_broadcast(&cond);
        barieraCount = 0;
    }
    pthread_mutex_unlock(&lock);
}
```

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

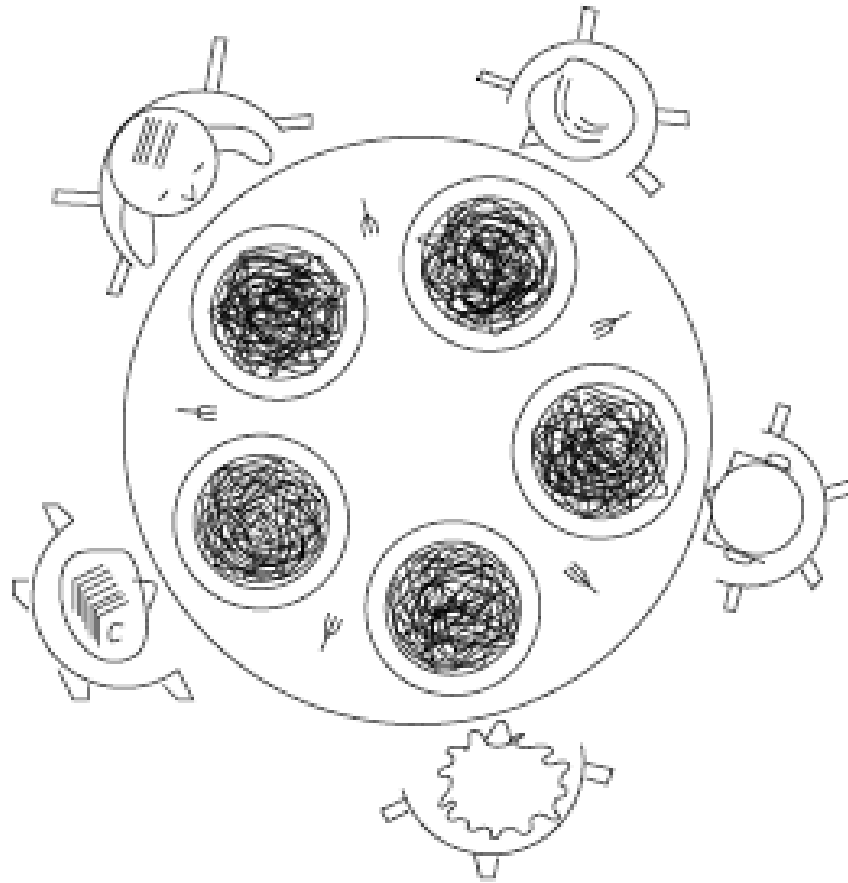
- Může dojít k zablokování
- Může dojít k vyhladovění

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

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

Očísľujeme **filozofy**: 0 .. 4

Očísľujeme **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;  
void filosof(i: integer)  
{  
    premyslej();  
    zvedni (i);  
    zvedni ((i+1) mod N);  
    jez();  
    poloz(i);  
    poloz((i+1) mod N)  
};
```

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

1. Filozof zvedne levou vidličku a zjistí, zda je pravá vidlička dostupná.
2. Pokud není, položí levou, počká nějakou dobu a zkusí celý postup 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

Správné 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
```

```
int f[0 .. N-1];
```

```
// počet vidlicek dostupných filozofovi
```

```
int a[0 .. N-1];
```

```
// podmínka vidličky k dispozici
```

```
int i;
```

```
void chci_jist ( int i )
```

```
{
```

```
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
```

```
};
```

```

void mam_dost ( int i)
{
    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 levý soused 2 vidličky?
        a [ (i-1) mod N] . signal;
    if f [ (i+1) mod N] == 2 then             // má pravý soused 2 vidličky?
        a [ (i+1) mod N] . signal;
};

begin                                         // inicializace monitoru, filozof má 2 vidličky
    for i=0 to 4 do
        f[i] = 2
    end;

```