Komunikace a synchronizace procesů

PB 152 ◇ Operační systémy

Jan Staudek
http://www.fi.muni.cz/usr/staudek/vyuka/

Verze: jaro 2017

Motto platné již 35 let

Designing correct routines for controlling concurrent activities proved to be one of the most difficult aspects of systems programming.

The ad hoc techniques used by programmers of early multiprogramming and real-time systems were always vulnerable to subtle programming errors whose effects could be observed only when certain relatively rare sequences of actions occurred.

The errors are particularly difficult to locate, since the precise conditions under which they appear are very hard to reproduce.

THE COMPUTER SCIENCE AND ENGINEERING RESEARCH STUDY , MIT Press, 1980

Osnova přednášky

- potřeba a formy IPC (*Interprocess Communication*)
- IPC sdílenou pamětí
- problémy synchronizace (*race conditions*)
- problém kritické sekce
- řešení problému kritické sekce softwarově na úrovni aplikace
- řešení problému kritické sekce speciálními instrukcemi
- semafory
- IPC výměnou zpráv
- klasické synchronizační úlohy řešené pomocí semaforů
- monitory
- příklady synchronizace z konkrétních OS

Potřeba a formy IPC, aktivity se dějí souběžně

- Multi-threading
  - ✓ souběžně existující vlákna sdílejí adresový prostor
- Multi-programming, multi-tasking
  - ✓ souběžně existující procesy a vlákna jsou střídavě realizované 1 nebo více procesory
- Multi-processing
  - ✓ participace více procesorů na multi-taskingu
- Distribuované zpracování
  - ✓ souběžně existující procesy jsou realizované více uzly sítě
- Souběžné aktivity mohou mezi sebou soupeřit o omezené zdroje (periferie, soubory dat, oblasti paměti, . . .)
- Souběžné aktivity mohou mezi sebou komunikovat výměnou zpráv
- Souběžné aktivity mohou svoje běhy vzájemně synchronizovat

Formy koexistence – soupeření souběžných aktivit

- souběžné procesy (vlákna) potřebují speciální podporu od OS
  - ✓ pro komunikace mezi sebou výměnou zpráv /sdílením paměti
  - ✓ pro přidělování procesoru a dalších zdrojů pro jejich běh
  - ✓ pro vzájemnou synchronizaci svých běhů
- soupeření – první ze dvou forem koexistence procesů / vláken
  - ✓ souběžné procesy se ucházejí o zdroje – procesor, FAP, globálně dostupné periferie, soubory dat, . . .
  - ✓ zdroje soupeřícím procesům typicky přiděluje OS
  - ✓ OS efektivně isoluje soupeřící procesy, aby se chybně neovlivňovaly
  - ✓ soupeřící procesy se vzájemně neznají, soupeřící proces si není vědom existence ostatních soupeřících procesů
  - ✓ realizace procesů musí být deterministická, reprodukovatelná, procesy musí být rušitelné a restartovatelné bez bočních efektů

Formy koexistence – Kooperace souběžných aktivit

- kooperace – druhá forma koexistence procesů / vláken
  - ✓ kooperující procesy sdílí jistou množinu zdrojů, vzájemně se znají
  - ✓ kooperace se dosahuje buďto implicitním sdílením zdrojů nebo explicitní komunikací kooperujících procesů
  - ✓ vlákna jednoho procesu obvykle kooperují, nesoupeří
  - ✓ procesy mohou jak kooperovat, tak i soupeřit
- proč vlákna/procesy kooperují
  - ✓ aby mohly sdílet jisté zdroje
  - ✓ aby se mohly nezávislé akce řešit souběžně, např. čtení příštího bloku dat během zpracovávání již přečteného bloku dat
  - ✓ aby se podporovala modulárnost architektury aplikačního systému

```
cat infile | tr ' ' '\012' |tr '[A-Z]' '[a-z]' | sort | uniq -c
```


Přínosy kooperace a sdílení

- možnost sdílet zdroje, informace eliminuje nutnost redundance
- dojde k urychlení výpočtu prováděného po částech paralelně
- modularizace, jednotlivé systémové funkce lze řešit samostatnými procesy či vlákny
- pohodlí, i uživatel jednotlivec může souběžně řešit více úkolů (editace, tisk, kompilace, . . .)

Příklad problému nekonzistence při souběžnosti

- Souběžný přístup ke sdíleným údajům se musí mnohdy provádět neatomickými operacemi
  - ✓ Udržování konzistence dat požaduje používání mechanismů, které zajistí deterministické provádění akcí kooperujících procesů
- Příklad neatomické operace nad sdílenými proměnnými

```
void echo()
{
  chin = getchar();
  chout = chin;
  putchar(chout);
}
```

  - ✓ procesy P_1 a P_2 provádějí tutéž proceduru ***echo*** a
  - ✓ operují se sdílenými proměnnými ***chin***, ***chout***
  - ✓ oba procesy lze přerušit ve kterémkoliv místě
  - ✓ o rychlosti postupu každého z procesů nelze nic předpovědět

Příklad problému nekonzistence při souběžnosti

- Příklad možného průběhu procesů P1 a P2

P1	P2
chin = getchar();	...
chout = chin;	...
putchar(chout);	...
...	chin = getchar();
...	chout = chin;
...	putchar(chout);
...	...

  - ✓ Průběh je validní
  - ✓ V multitaskingovém systému však nemůžeme nic předpokládat o rychlosti běhů jednotlivých procesů

    neřízená kooperace je zdrojem časové závislých chyb

Příklad problému nekonzistence při souběžnosti

- Příklad jiného možného průběhu procesů P1 a P2

P1	P2
...	...
chin = getchar();	...
...	chin = getchar();
chout = chin;	...
...	chout = chin;
putchar(chout);	...
...	putchar(chout);
...	...

  - ✓ Znak načtený v P1 se ztrácí dříve než je zobrazený
  - ✓ Znak načtený v P2 se vypisuje v P1 i P2

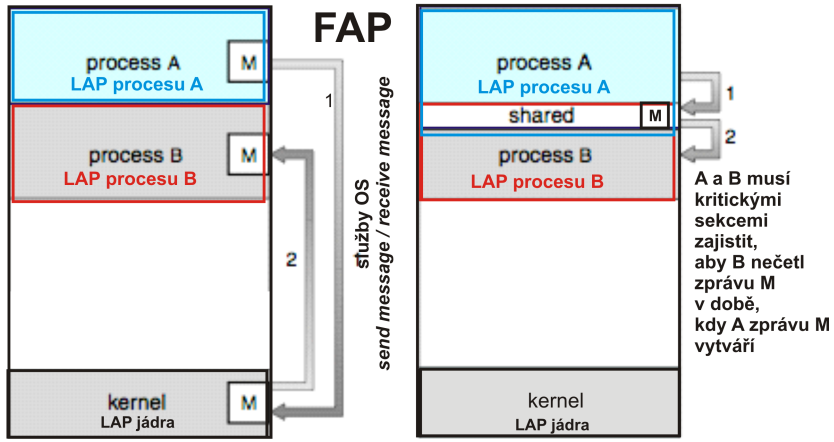
Typové úlohy související se souběžností

- Synchronizace procesů – čekání procesu na událost
- Komunikace mezi procesy – výměna zpráv
  - ✓ rozšíření synchronizace pro koordinaci různých aktivit, ke sdělení o vzniku události se přidává sdělovaná informace – zpráva
- Sdílení prostředků – soupeření (*race condition*)
  - ✓ procesy používají a modifikují sdílená data, operace zápisu těchto dat musí být vzájemně vyloučené, operace zápisu těchto dat musí být vzájemně vyloučené s operacemi jejich čtení, operace jejich čtení být realizovány souběžně
  - ✓ Pro zabezpečení integrity dat musí programátor použít tzv. kritické sekce zajišťující serializaci konfliktních operací (***write x write, read x write***)
- Může docházet k „uváznutí" – každý proces v systému čeká na událost či zprávu generovanou v některém jiném procesu v systém nebo na uvolnění vstupu do kritické sekce

Bázové formy komunikace mezi procesy

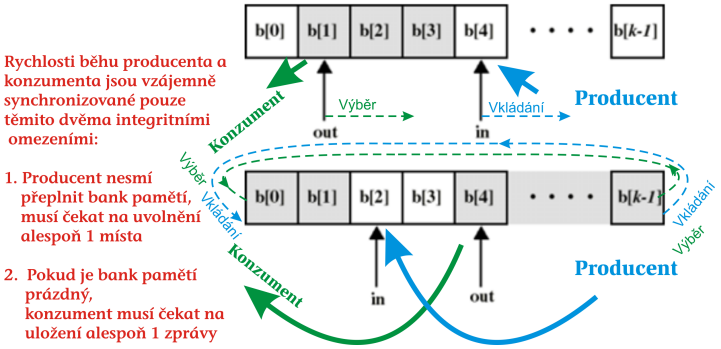
- komunikace mezi procesy – IPC, *Interprocess Communication*
- Formy IPC
  - ✓ sdílená paměť, *shared memory*
  - ✓ výměna zpráv, *message passing*

Sdílená paměť, výměna zpráv



Příklad: sdílená vyrovnávací paměť s omezenou kapacitou

- Použití sdílené vyrovnávací paměti s omezenou kapacitou pro výměnu dat mezi 2 procesy, producentem a konzumentem.
  Častý název úlohy: Producent/Konzument (zpráv), resp. také „*Bounded-Buffer problem*“



Program producenta a program konzumenta

Sdílená data:

```
#define BUFFER_SIZE 10
typedef struct { ... } item;
item buffer[BUFFER_SIZE];
int in = 0; int out = 0; int count = 0;
```

Producent

```
item nextProduced;

while (1) { ...          /* produkce */
  while (count == BUFFER_SIZE)
    ;                    /* do nothing */
  ++count;
  buffer[in] = nextProduced;
  in = (in + 1) % BUFFER_SIZE;
}
```

Konzument:

```
item nextConsumed;

while (1) {
  while (count == 0)
    ;                    /* do nothing */
  - - count;
  nextConsumed = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  ...                    /* konzumace */ }
```


Nutná synchronizace – Race Condition

- ✓ Souběh R a W neatomickými operacemi stejné položky dat, např. příkazy $++count$ a $--count$ se musí provádět atomicky
- ✓ provést se atomicky ≡ provést se bez přerušení

- příkaz $++count$
  bude ve strojovém jazyku implementovaný takto:
  $register_1 = count$
  $register_1 = register_1 + 1$
  $count = register_1$
- příkaz $--count$
  bude ve strojovém jazyku implementovaný takto:
  $register_2 = count$
  $register_2 = register_2 - 1$
  $count = register_2$

Nutná synchronizace – Race Condition, 2

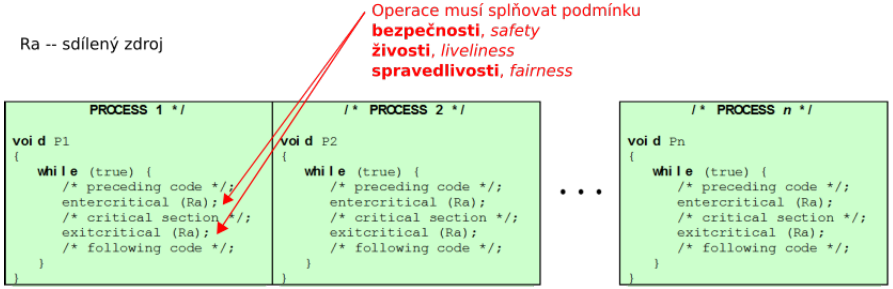
- Jestliže se producent i konzument pokusí zkorigovat vyrovnávací paměť současně, mohou se interpretace instrukcí jejich programů v čase prokládat – konkrétní prokládání je ale nepredikovatelné
- Příklad:
  - ✓ Nechť $count$ má iniciální hodnotu 5.
  - ✓ Provedení operací $++count$; a $--count$; nesmí hodnotu $count$ změnit
  - ✓ Možné proložení operací $++count$; a $--count$;:
    $producent : register_1 = count \quad (register_1 = 5)$
    $producent : register_1 = register_1 + 1 \quad (register_1 = 6)$
    $konzument : register_2 = count \quad (register_2 = 5)$
    $konzument : register_2 = register_2 - 1 \quad (register_2 = 4)$
    $producent : count = register_1 \quad (count = 6)$
    $konzument : count = register_2 \quad (count = 4)$
  - ✓ Hodnota $count$ je 4, ne správná hodnota 5.

Problém kritické sekce

- n procesů soupeří o právo používat jistý sdílený zdroj vzájemně výlučně
- v každém procesu z n se nachází segment kódu programu nazývaný kritická sekce, ve kterém proces vzájemně výlučně přistupuje ke sdílenému zdroji
- je potřeba zajistit, že v jisté kritické sekci sdružené s jistým zdrojem, se bude nacházet nejvýše jeden proces
- modelové prostředí pro hledání řešení problému kritické sekce
  - ✓ předpokládá se, že každý proces běží nenulovou rychlostí
  - ✓ nic se nepředpokládá o relativní rychlosti procesů
  - ✓ žádný proces nezůstane v kritické sekci nekonečně dlouho

Ilustrace vzájemného vyloučení

- Rychlost běhu procesů neznáme



Vlastnosti správného řešení problému kritické sekce

- Dosažení vzájemného vyloučení, podmínka bezpečnosti, ,,*safety*"
  - ✓ Jestliže některý proces provádí svoji kritickou sekci, žádný jiný proces nemůže provádět svoji kritickou sekci sdruženou se stejným zdrojem
- Trvalost postupu, podmínka živosti, ,,*liveliness*", *„progress"*
  - ✓ Jestliže žádný proces neprovádí svoji kritickou sekci sdruženou s jistým zdrojem a existuje alespoň jeden proces, který si přeje vstoupit do kritické sekce sdružené se tímto zdrojem, pak výběr procesu, který do takové kritické sekce vstoupí, se nesmí odkládat
- Konečnost doby čekání, podmínka spravedlivosti, *„fairness"*
  - ✓ Po vydání žádosti jistého procesu z n procesů o vstup do jisté kritické sekce a před uspokojením tohoto požadavku může může být povolen vstup do sdružené kritické sekce nejvýše $n–1$ procesům

Další vlastnosti správného řešení problému kritické sekce

- Proces, který končí svoji činnost v okamžiku,
  kdy se nenachází v kritické sekci,
  musí tak učinit bez interference s ostatními procesy
- Nelze nic předpokládat o relativní rychlosti procesů a počtu procesorů
- Proces pobývá v kritické sekci konečnou dobu

Koncept řešení problému kritické seke

- pro názornost předpokládáme existenci 2 procesů, P_0 a P_1
- generická struktura procesu P_i

```
do {
    enteringCriticalSection()
    critical section
    leavingCriticalSection()
    reminder section
} while (1);
```

- procesy mohou za účelem dosažení synchronizace svých akcí sdílet společné proměnné
- počátečně připustíme činné (aktivní) čekání procesu na splnění podmínek pro vstup do kritické sekce v *enteringCriticalSection()* – busy waiting

Elementární řešení problému KS – maskování přerušení

- enteringCriticalSection() = disable_interrupt
  leavingCriticalSection() = enable_interrupt
- proces, který první provede vstup do kritické sekce, zamaskuje přerušení na procesoru
- vlastnosti:
  - ✓ uplatnitelné pouze v 1-procesorových systémech
  - ✓ hloupé, neefektivní řešení
  - ✓ plná eliminace rysů multiprogramování
  - ✓ kritická sekce se stává nedělitelným blokem fyzicky, nikoli logicky

Možné kategorie základen pro řešení problému KS

- Softwarové řešení nezprostředkovávané jinými službami
  - ✓ algoritmy, jejichž správnost se nespoléhá na žádné další služby
  - ✓ používají standardní instrukční repertoár (LOAD, STORE, ...)
  - ✓ na možnost vstupu do KS aktivně čekají, *busy waiting*
- Hardwarové řešení nezprostředkovávané jinými službami
  - ✓ algoritmy, jejichž správnost se nespoléhá na žádné další služby
  - ✓ používají speciální instrukce strojového jazyka (TST, XCHG, ...)
  - ✓ na možnost vstupu do KS aktivně čekají, *busy waiting*
- Softwarové řešení zprostředkované operačním systémem
  - ✓ potřebné služby a datové struktury poskytuje OS
  - ✓ na možnost vstupu do KS se čeká pasivně, ve frontě
  - ✓ ex. podpora volání služeb v programovacích systémech/jazycích – semafory, monitory, zasílání zpráv

Čistě softwarové řešení

```
boolean   flag [2];

void   P0()
{
        while   (true) {
                flag [0] = true;

                while ( flag [1]                ) /* do nothing */  ;
                /*  critical section            */ ;
                flag [0] = false;
                /*  remainder       */ ;
        }
}
void   P1()
{
        while   (true) {
                flag [1] = true;

                while ( flag [0]                )   /* do nothing */  ;
                /*  critical section            */ ;
                flag [1] = false;
                /*  remainder       */
        }
}
void main()
{
        flag [0] = false;
        flag [1] = false;
        parbegin (    P0, P1);
}
```

Dosažení vzájemného vyloučení

Pole *flag*
oba procesy sdílí.

Jakmile P0 nastaví *flag[0]* na *true*,
P1 nemůže vstoupit do kritické sekce.

Jakmile P1 nastaví *flag[1]* na *true*,
P0 nemůže vstoupit do kritické sekce.

Co se stane, když oba procesy
současně nastaví svůj flag na true ?

UVÁZNOU

Čistě softwarové řešení, Petersonovo řešení

```
boolean   flag [2];
int   turn;
void   P0()
{
        while   (true) {
                flag [0] = true;
                turn = 1;
                while ( flag [1] &&  turn == 1) /* do nothing */  ;
                /*  critical section            */ ;
                flag [0] = false;
                /*  remainder       */ ;
        }
}
void   P1()
{
        while   (true) {
                flag [1] = true;
                turn = 0;
                while ( flag [0] && turn == 0)   /* do nothing */  ;
                /*  critical section            */ ;
                flag [1] = false;
                /*  remainder       */
        }
}
void main()
{
        flag [0] = false;
        flag [1] = false;
        parbegin (    P0, P1);
}
```

Dosažení vzájemného vyloučení

Pole *flag* a proměnnou *turn*
oba procesy sdílí.

Jakmile P0 nastaví *flag[0]* na *true*,
P1 nemůže vstoupit do kritické sekce.

Jakmile P1 nastaví *flag[1]* na *true*,
P0 nemůže vstoupit do kritické sekce.

Vzájemnému blokování zabraňuje
sdílená proměnná *turn*,
která nabude hodnoty 0 nebo 1
i v případě souběhu příkazů
turn = 1 a *turn = 0*.

Žádný proces nemůže usurpovat
kritickou sekci trvale, při výstupu
vždy dá šanci vstoupit do kritické
sekce procesu, se kterým soupeřil.

Čistě softwarové řešení, Petersonovo řešení

- Petersonovo řešení nesplňuje podmínku spravedlivosti
  - ✓ Vlákna neuváznou, některé vlákno může ale stárnout
    - při plně synchronním běhu vláken rozhoduje o vítězi soupeření náhodně určená hodnota proměnné *turn* (0 nebo 1)
- Petersonovo řešení lze generalizovat pro libovolný, předem známý počet procesů

Hardwarová podpora synchronizace, speciální instrukce

- Monoprocesory mohou problém násobnosti vstupu do KS vyřešit zamaskováním přerušení
  - ✓ v multiprocesorových systémech zamaskování přerušení na jednom procesoru problém neřeší
- Ex. speciální atomické (≡ nepřerušitelné) synchronizační instrukce vhodné i pro multiprocesory (x86, Sparc, IBM z series, Intel IA-32 (Pentium), IA-64 (Itanium) . . .)
  - ✓ nepřerušitelné – do hlavní paměti přistupují vícekrát, nepřerušitelně

Hardwarová podpora synchronizace, speciální instrukce

- Test-and-Set Lock, TSL REGISTER,LOCK –
  - ✓ získání hodnoty proměnné LOCK z FAP do registru a nastavení její nenulové hodnoty ve FAP, atomicky
- XCHG, XCHG REGISTER,LOCK –
  - ✓ výměna obsahu dvou paměťových míst (registr x buňka FAP a nebo příp. buňka FAP x buňka FAP) atomicky
- compare-and-swap (int *word, int testval, int newval)
  - ✓ testuje hodnotu proměnné (**word*) proti hodnotě *testval*, při shodě se nahradí hodnotu proměnné hodnotou *newval*; jinak ponechá původní hodnotu proměnné
  - ✓ vždy vrací původní hodnotu proměnné, takže místo v paměti se mění pokud vracená hodnota se shoduje s hodnotou použitou jako vzor testu (proběhne *swap*)

Test-and-Set Lock, princip použití

podprogramy řešící vstup a výstup do/z kritické sekce

```
enter_region:
    TSL REGISTER,LOCK      | copy lock to register and set lock to 1
    CMP REGISTER,#0        | was lock zero?
    JNE enter_region       | if it was nonzero, lock was set, so loop
    RET                    | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0           | store a 0 in lock
    RET                    | return to caller
```

vyjádření v programovacím jazyky vyšší úrovně

Nedělitelně provede x:=r a r:=1 — 1 značí obsazenost kritické sekce

x je lokální proměnná

r je globální registr iniciálně = 0 — 0 značí dostupnost kritické sekce

```
repeat (test&setloc(x)) until x = 0;   | Pokud bylo r=1, cykluje, busy waiting
  < critical section >                 | Pokud platilo r=0 a nyní platí r=1.
  r:= 0;                               | Kritická sekce se uvolňuje.
```


XCHG, princip použití paměť x registr

podprogramy řešící vstup a výstup do/z kritické sekce

```
enter_region:
    MOVE REGISTER,#1       | put a 1 in the register
    XCHG REGISTER,LOCK     | swap the contents of the register and lock variable
    CMP REGISTER,#0        | was lock zero?
    JNE enter_region       | if it was non zero, lock was set, so loop
    RET                    | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0           | store a 0 in lock
    RET                    | return to caller
```

vyjádření v programovacím jazyky vyšší úrovně

Nedělitelně vymění *x* a *r*

x je lokální proměnná

r je globální registr iniciálně = 1 — 1 značí dostupnost kritické sekce

```
x := 0;
repeat (xchg(x, r)) until x = 1;   | příprava zamykací hodnoty
  < critical section >             | Cykluje pokud r=0,
  xchg(x, r);                      | Bylo r=1 nyní je r=0 a x=1.
                                   | Uvolnění kritické sekce, r=1.
```


XCHG, princip použití paměť x paměť

```
/* program mutualexclusion */
int const n = /* number of processes*/;
int bolt;
void P(int i)
{
  while (true) {
    int keyi = 1;
    do exchange (&keyi, &bolt)
    while (keyi != 0);
    /* critical section */;
    bolt = 0;
    /* remainder */;
  }
}
void main()
{
  bolt = 0;
  parbegin (P(1), P(2), . . ., P(n));
}
```

bolt:
synchronizační
proměnná

= 0 volno

compare-and-swap, princip použití

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
  while (true) {
    while (compare_and_swap(&bolt, 0, 1) == 1)
        /* do nothing */;
    /* critical section */;
    bolt = 0;
    /* remainder */;
  }
}
void main()
{
  bolt = 0;
  parbegin (P(1), P(2), . . . ,P(n));

}
```


Závěry z prvních dvou způsobů řešení

- Negativa softwarového řešení
  - ✓ Procesy, které žádají o vstup do svých KS, to dělají metodou „busy waiting“, spotřebovávají čas procesoru
  - ✓ Není splněna podmínka spravedlnosti
- závěry ke speciálním instrukcím
  - ✓ klady:
    – vhodné i pro multiprocesory
    – na rozdíl od prostého zamaskování / odmaskování přerušení
  - ✓ negativa
    – aktivní čekání
    – možnost stárnutí – díky náhodnosti řešení konfliktu
    – možnost uváznutí – díky činnému čekání na vstup do kritické sekce
    – řešení nesplňuje podmínku spravedlnosti
- negativa nevadí, pokud jsou KS krátké a volané řídce
  - ✓ použití v jádru OS toto omezení splňuje

Semafory

- Synchronizační nástroj
  - ✓ proměnná typu *semaphore* nabývající hodnot
    – *volno* (podle způsobu implementace: zvednutý semafor, 1, true, . . .)
    – *obsazeno* (shozený semafor, 0, false, . . .)
  - ✓ operace *oznamuji událost* (zvedám semafor, uvolňuji cestu)
    – časté názvy: ***release***, ***signal***, ***semSignal***, ***V*** (z NL – Vrhogen),
  - ✓ operace *čekám na událost* (čekám na zvednutý semaforu a shazuji ho)
    – detekcí události se informace o vzniku události ztrácí
    – vyžaduje se např. zajištění volné cesty do kritické sekce
    – časté názvy: ***acquire***, ***wait***, ***semWait***, ***P***, (z NL – Proberen),
  - ✓ operace *inicializace* semaforu na počáteční hodnotu
- Semafory jsou službou poskytovanou procesům/vláknům implementovanou v nižší vrstvě software vůči procesům/vláknům (aplikacím)

Semafory

- Požaduje se, aby se čekání na událost realizovalo pasivně
  - ✓ Podpůrná vrstva potlačí běh procesu/vlákna do doby vzniku události
  - ✓ Standardně se semafory implementují jako služba jádra OS
  - ✓ Jádro OS poskytuje službu nad identifikovatelným semaforem, aplikační účel použití konkrétního semaforu si definuje aplikace
- Nechť je v semaforu S volno/obsazeno implementované 1/0 iniciální hodnotou je 1
- Pak lze operace nad semaforem S (ATOMICKÉ VŮČI S) symbolicky vyjádřit následovně (zatím s aktivním čekáním)

```
acquire(S) {
    while S ≤ 0; // no-operation
 S – –;
}
```

```
release(S) {
    S++;
}
```

Vzájemné vyloučení KS pomocí binárního semaforu

```
Semaphore S;          % inicializovaný na 1

...
acquire(S);
   criticalSection();
release(S);
...
```


Implementace semaforu jako služba z rozhraní služeb OS

- Jsou potřeba dvě pomocné operace, vnitřní operace v jádru
  - ✓ *block* – potlačuje proces, který operaci *acquire(S)* vyvolal (*běžící* proces dá mezi procesy *čekající na semafor*)
  - ✓ *wakeup(P)* – přeřazuje čekající proces P mezi připravené procesy
- Idea implementace operací *acquire(S)* a *release(S)* v jádru

```
acquire(S){
   S.value − −;
   if (S.value < 0) {
      block;
   }
}
```

```
release(S){
   S.value++;
   if (value ≤ 0) {
     ... // z fronty čekajících
     ... // je odebrán process P
      wakeup(P);
   }
}
```


Implementace semaforu, 2

- Implementace musí zaručit, že žádné dva procesy nemohou provádět operace ***acquire***() a/nebo ***release***() nad stejným semaforem současně
- splnění podmínkek bezpečnosti, živosti a spravedlnosti vůči žádajícím procesům je problém řešený softwarově v jádru OS
  - ✓ živost a spravedlnost zajistí implementace filozofie FIFO v operacích *block* a *wakeup(P)*
  - ✓ bezpečnost – vzájemná výlučnost je dosažitelná snadno

Implementace semaforu, 3

- Častým řešením operace ***release***() je přesunutí procesů čekajících na zvednutí semaforu mezi připravené procesy s nastavením čítače instrukcí na zopakování operace ***acquire***()
  - ✓ pořadí aktivace procesů pak určuje dispečer
- V některých implementacích semaforů mohou jejich hodnoty nabývat i záporných hodnot, které pak vesměs vyjadřují počet čekajících aktérů na zvednutí semaforu

Implementace semaforu, 4

- Implementace semaforu se stává problémem kritické sekce
- operace ***acquire***() a ***release***() musí být atomické
- na 1 procesorovém stroji lze zajistit atomicitu zamaskováním přerušení v jádru OS
- na multiprocesoru se musí použít buďto přímé softwarové řešení kritické sekce nebo se využijí speciální instrukce, pokud je procesor podporuje
  - ✓ ,,busy waitnig"nelze plně eliminovat, lze ho přesunout z aplikační úrovně (kde mohou být kritické sekce dlouhé) do úrovně jádra OS pro implementaci atomicity operací ***acquire***() a ***release***()
- v distribuovaném prostředí se musí použít speciální distribuované algoritmy, viz předmět PA 150

Ilustrace implementace operací na semaforem

```
semWait(s)
{
   while (compare_and_swap(s.flag, 0 , 1) == 1)
      /* do nothing */;
   s.count--;
   if (s.count < 0) {
      /* place this process in s.queue*/;
      /* block this process (must also set s.flag to 0)*/;

   }
   s.flag = 0;
}

semSignal(s)
{
   while (compare_and_swap(s.flag, 0 , 1) == 1)
      /* do nothing */;
   s.count++;
   if (s.count <= 0) {
      /* remove a process P from s.queue */;          */;
      /* place process P on ready list */;
   }
   s.flag = 0;
}
```

Kritická sekce

Kritická sekce

Semafor coby synchronizátor

- Má se provést akce B v procesu P_j až po té, co se provede akce A v procesu P_i
- použije se semafor ***flag*** inicializovaný na 0
  - ✓ 0 – událost nenastala, 1 – událost nastala

P_i:	P_j:
...	...
A	*acquire(flag)*
release(flag)	B

Uváznutí a stárnutí

- Uváznutí
  - ✓ dva nebo více procesů neomezeně dlouho čekají na událost, kterou může generovat pouze jeden z čekající procesů
  - ✓ Nechť S a Q jsou dva semafory inicializované na 1 a řádky vyjadřují tok času

P_i:	P_j:
acquire(S);	*acquire(Q);*
acquire(Q);	*acquire(S);*
...	...
release(S);	*release(Q);*
release(Q);	*release(S);*

- Stárnutí
  - ✓ neomezené blokování, proces nemusí být odstraněný z fronty na semafor nikdy (předbíháním procesy s vyššími prioritami, ...)

Semafor, typy

- Binární semafor (také – mutex lock)
  - ✓ nabývá celočíselných hodnot z intervalu $< 0, 1 >$
  - ✓ odpovídá interpretaci obsazeno/ volno
  - ✓ lze implementovat i instrukcemi TSL nebo XCHG
  - ✓ lze implementovat i přímým Petersonovým řešením
- Obecný semafor
  - ✓ celočíselná hodnota z intervalu $< 0, n >$, $n > 1$
  - ✓ slouží např. k čítání událostí apod.
  - ✓ lze implementovat pouze pomocí komplexnějších funkcí jádra
- implementovatelnost obecného semaforu
  - ✓ binární semafor lze snadno implementovat
  - ✓ obecný semafor lze implementovat semaforem binárním, viz dále

Implementace obecného semaforu

- Datové struktury obecného semaforu S s maximální hodnotou C
  - ✓ *binary-semaphore S1, S2; int C;*
- Inicializace:
  - ✓ *S1 = 1; S2 = 0; C = iniciální hodnota semaforu S;*
- operace acquire: a release:

```
acquire(S1);
C--;
if (C < 0) {
    release(S1);
    acquire(S2);
}
release(S1);
```

```
acquire(S1);
C++;
if (C ≤ 0)
    release(S2);
release(S1);
```


Klasické synchronizační úlohy

- Producent – konzument (*Bounded-Buffer Problem*)
  - ✓ předávání zpráv mezi 2 procesy
- Čtenáři a písaři (*Readers and Writers Problem*)
  - ✓ souběžnost čtení a modifikace dat (v databázi, . . .)
- Úloha o večeřících filozofech
  - ✓ ilustračně zajímavý problém pro řešení uváznutí

– 5 filozofů buď myslí nebo jí
– jí špagety jen 2 vidličkami
– co se stane, když se všech 5 filozofů najednou chopí např. levé vidličky ?
– „no přece časem zemřou hladem, děti“

Řešení úlohy producent – konzument

- Potřebujeme 1 binární semafor pro vzájemné vyloučení operací s bankem bufferů – $mutex$
  - ✓ k banku bufferů smí přistoupit najednou jediný proces
- Potřebujeme 2 obecné semafory pro synchronizaci producenta a konzumenta stavem banku N bufferů –
  – obecný semafor pro indikaci počtu plných ($full$) a
  – obecný semafor pro indikaci počtu prázdných ($empty$)
  - ✓ co nebylo produkováno, nelze konzumovat, konzumovat lze jen když platí $full > 0$
  - ✓ nelze produkovat do plného banku bufferů, produkovat lze jen když platí $empty > 0$,

Řešení úlohy producent – konzument, 2

- Sdílené datové struktury:
  - ✓ *semaphore full, empty, mutex;*
- Inicializace:
  - ✓ *full = 0; empty = N; mutex = 1*
- operace producenta: a konzumenta:

do {	*do {*
vytvoř data v nextp;	*acquire(full);*
acquire(empty);	*acquire(mutex);*
acquire(mutex);	*odeber data z bufferu do nextc;*
přidej nextp do buferu;	*release(mutex);*
release(mutex);	*release(empty);*
release(full)	*konzumuj data z nextc*
} while (true);	*} while (true);*

Variace na úlohu producent konzument

- Mějme dva vícevláknové procesy A a B
- Každé vlákno běží cyklicky a vyměňuje si zprávu s některým vláknem druhého procesu
- Zprávou je např. číslo ukládané do sdíleného bufferu
- Musí platit
  - ✓ Poté co vlákno z A zpřístupní zprávu některému vláknu z B, může pokračovat v běhu až získá zprávu od tohoto vlákna z B
  - ✓ Poté co vlákno z B zpřístupní zprávu některému vláknu z A, může pokračovat v běhu až získá zprávu od tohoto vlákna z A
  - ✓ Jakmile vlákno z A zprávu zpřístupní, musí zajistit, aby ji jiné vlákno z A nepřepsalo, dokud si ji některé vlákno z B nepřevezme
  - ✓ Jakmile vlákno z B zprávu zpřístupní, musí zajistit, aby ji jiné vlákno z B nepřepsalo, dokud si ji některé vlákno z A nepřevezme

Variace na úlohu producent konzument

```
semaphore notFull_A = 1, notFull_B = 1;
semaphore notEmpty_A = 0, notEmpty_B = 0;
int buf_a, buf_b;
```

```
thread_A(...)
{
   int var_a;
   ...
   while (true) {
      . . .
      var_a =...;
      semWait(notFull_A);
         buf_a = var_a;
         semSignal(notEmpty_A);
      semWait(notEmpty_B);
         var_a = buf_b;
         semSignal(notFull_B);
      . . .;
   }
}
```

```
thread_B(...)
{
   int var_b;
   ...
   while (true) {
      . . .
      var_b =...;
      semWait(notFull_B);
         buf_b = var_b;
         semSignal(notEmpty_B);
      semWait(notEmpty_A);
         var_b = buf_a;
         semSignal(notFull_A);
      . . .;
   }
}
```


Čtenáři a písaři

- operace zápisu do sdíleného zdroje musí být exklusivní, vzájemně vyloučené s jakoukoli jinou operací
- operace čtení mohou čtený zdroj sdílet
- libovolný počet procesů–čtenářů může číst jeden a tentýž zdroj současně
- v jednom okamžiku smí daný zdroj modifikovat pouze jeden proces–písař
- jestliže proces–písař modifikuje zdroj, nesmí ho současně číst žádný proces–čtenář
- čtenář není konzument, písař není producent, jde o jinou úlohu

Čtenáři a písaři, 2

- Čtenáři a písaři s prioritou čtenářů
  - ✓ první čtenář přistupující ke zdroji zablokuje všechny písaře,
  - ✓ poslední z čtenářů končící čtení zdroje uvolní přístup ke zdroji případně čekajícím písařům, písaři se vzájemně vylučují
  - ✓ písaři mohou stárnout, pokud bude trvale alespoň 1 čtenář přistupovat ke zdroji
- Čtenáři a písaři s prioritou písařů
  - ✓ první čtenář přistupující ke zdroji zablokuje všechny písaře,
  - ✓ poslední z čtenářů končící čtení zdroje uvolní přístup ke zdroji případně čekajícím písařům
  - ✓ první písař žádající vstup do kritické sekce novým čtenářům přístup ke zdroji zakáže, noví čtenáři musí čekat na pasivitu všech písařů
  - ✓ čtenáři mohou stárnout, pokud bude ve frontě trvale alespoň 1 písař

Čtenáři a písaři s prioritou čtenářů

Sdílené datové struktury:

✓ *semaphore wrt, readcountmutex; var readcount;*

Inicializace:

✓ *wrt = 1; readcountmutex = 1; readcount = 0;*

písař:

```
acquire(wrt);
...
písař modifikuje zdroj
...
release(wrt);
```

a

čtenář:

```
acquire(readcountmutex);
   readcount++;
   if (readcount == 1) acquire(wrt);
release(readcountmutex);
 ... čtení sdíleného zdroje
acquire(readcountmutex);
   readcount – –;
   if (readcount == 0) release(wrt);
release(readcountmutex);
```


Čtenáři a písaři s prioritou písařů

Sdílené datové struktury:

✓ *semaphore wrt, rdr, rqueue, writecountmutex, readcountmutex; var readcount, writecount;*

Inicializace:

✓
```
wrt = 1;      % zajištění vzájemné vyloučení w-w / r-w
rdr = 1;      % blokování čtenářů chce-li zapisovat alespoň 1 písař
rqueue = 1;   % obecný semafor pro frontování nových čtenářů
writecountmutex = 1;
readcountmutex = 1;
readcount = 0;
writecount = 0;
```


Čtenáři a písaři s prioritou písařů, 2

písař:

```
acquire(writecountmutex);
    writecount ++;
    if (writecount == 1) acquire(rdr); % první z písařů zablokuje nové čtenáře
release(writecountmutex);
acquire(wrt);                          % vzájemné vyloučení písařů
    ... písař modifikuje zdroj ...
release(wrt);
acquire(writecountmutex);
    writecount – –;
    if (writecount == 0) release(rdr); % poslední z písařů odblokuje nové čtenáře
release(writecountmutex);
```

Čtenáři a písaři s prioritou písařů, 3

čtenář:

```
acquire(rqueue); % frontování dalších nových čt., pokud je alespoň 1 nový čt. blokovaný pís.
acquire(rdr);      % první písař zablokuje nového čtenáře
acquire(readcountmutex);
    readcount ++;
    if (readcount == 1) acquire(wrt);  % písaři musí vyčkat
release(readcountmutex);               % na dokončení rozpracovaných čtenářů
release(rdr);      % čištění informací o čekajících čtenářích
release(rqueue);   % čištění informací o čekajících čtenářích
... čtení sdíleného zdroje ...
acquire(readcountmutex);
    readcount --;
    if (readcount == 0) release(wrt);  % poslední rozpracovaný čtenář připouští písaře
release(readcountmutex);
```


Problémy se semafory

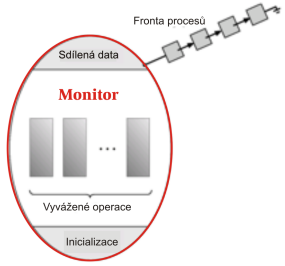
- semafory jsou mocný nástroj pro dosažení vzájemného vyloučení a koordinaci procesů
- operace $acquire(S)$ a $release(S)$ jsou prováděny více procesy a jejich účinek nemusí být vždy explicitně zřejmý
- semafor s explicitním ovládáním operacemi $acquire(S)$ a $release(S)$ je synchronizační nástroj nízké úrovně
- Chybné použití semaforu v jednom procesu hroutí souhru všech kooperujících procesů
- příklady patologického použití semaforů:

$acquire(S)$;	$acquire(S)$;	$release(S)$;
...	...	...
$acquire(S)$;	$release(T)$;	$acquire(S)$;

Monitory

- Monitor je synchronizační nástroj vysoké úrovně, umožňuje bezpečné sdílení nějakého zdroje souběžnými procesy/vlákny pomocí vyvážených procedur

```
monitor monitor-name {
        ... % deklarace proměnných
        public entry P1(...) { ... }
        public entry P2(...) { ... }
}
```

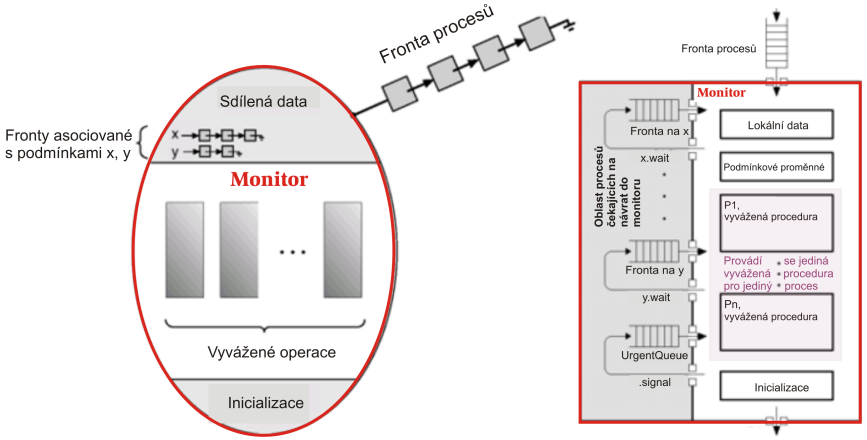


- Provádění vyvážených procedur P1, P2, ... se implicitně vzájemně vylučují (zajišťuje monitor)
- Podporují jazyky typu Java, prostředí .NET, ...

Monitory, podmínkové proměnné

- aby proces mohl čekat uvnitř provádění procedury monitoru, musí se v monitoru deklarovat proměnná typu *condition*, $condition\ x, y;$
- pro typ $condition$ jsou definovány dvě operace
  - ✓ *x.wait();*
    proces, který vyvolá tuto operaci je potlačen (a uvolní monitor) až do doby, kdy jiný proces provede operaci $x.signal$
  - ✓ *x.signal();*
    Splnění podmínky x signalizuje proces běžící v monitoru provedením operace $x.signal$.
    Operace $x.signal$ aktivuje právě jeden proces, který posléze znovu vstoupí do monitoru, až bude monitor volný.
    Pokud žádný proces nečeká na splnění podmínky x, je její provedení prázdnou operací.

Monitory, podmínkové proměnné, 2



- ✓ V monitoru se smí nacházet nejvýše 1 proces – typické řešení:
  signalizující proces bezprostředně opustí monitor
  a čeká na pokračování v běhu v monitoru ve frontě ***urgentqueue***

Producent-konzument pomocí monitorů, programy P a K

```
void producer()
{
     char x;
     while (true) {
     produce(x);
     append(x);
     }
}
void consumer()
{
     char x;
     while (true) {
       take(x);
       consume(x);
     }
}
void main()
{
     parbegin (producer, consumer);
}
```


Producent-konzument pomocí monitorů, deklarace monitoru

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];                                    /* space for N items */
int nextin, nextout;                                  /* buffer pointers */
int count;                                  /* number of items in buffer */
cond notfull, notempty;          /* condition variables for synchronization */

void append (char x)
{
     if (count == N) cwait(notfull);      /* buffer is full; avoid overflow */
     buffer[nextin] = x;
     nextin = (nextin + 1) % N;
     count++;
     /* one more item in buffer */
     csignal(notempty);                       /* resume any waiting consumer */
}
void take (char x)
{
     if (count == 0) cwait(notempty);   /* buffer is empty; avoid underflow */
     x = buffer[nextout];
     nextout = (nextout + 1) % N;
     count--;                                 /* one fewer item in buffer */
     csignal(notfull);                       /* resume any waiting producer */
}
{                                                          /* monitor body */
     nextin = 0; nextout = 0; count = 0;          /* buffer initially empty */
}
```

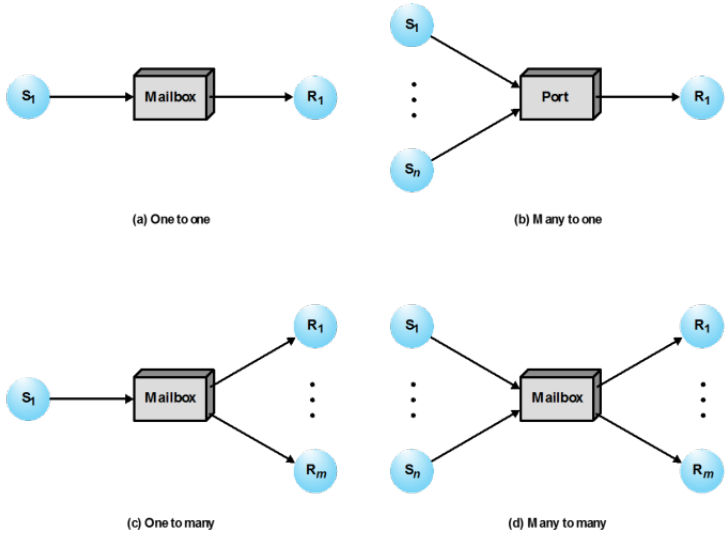

Výměna zpráv

- ☐ *send (destination, message)*, *receive (source, message)*
- ☐ Synchronizace
  - ✓ Blocking send, blocking receive, *randezvous*
  - ✓ Nonblocking send, blocking receive
  - ✓ Nonblocking send, Nonblocking receive
  - ✓ blocking = čeká se na komplementární operaci, synchronní
  - ✓ Nonblocking = nečeká se na komplementární operaci, asynchronní
- ☐ Adresování
  - ✓ přímé – udání identifikace cíle
  - ✓ nepřímé – udání místa pro zprávy odebírané přijímačem,
    port, *mailbox*
- ☐ Vztah mezi vysílačem a přijímačem při nepřímém adresování
  - ✓ 1:1, 1:n, m:1, m:n

Mailboxy a porty

- mailbox – schránka pro předávání zpráv
  - ✓ může být privátní pro dvojici komunikujících procesů
  - ✓ může být sdílená více procesy
  - ✓ OS může dělat typovou kontrolu zpráv
  - ✓ OS vytváří mailbox na pokyn procesu
  - ✓ proces je vlastník schránky, může ji rušit
  - ✓ schránka zaniká když její vlastník končí
- Port
  - ✓ mailbox patřící jednomu přijímacímu procesu a více procesům zasílajících zprávy
  - ✓ port vytváří přijímací proces
  - ✓ v modelu klient/server je přijímacím procesem server
  - ✓ port se ruší ukončením přijímacího procesu

Mailboxy a porty



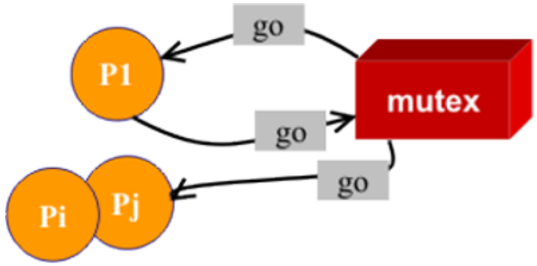
Vzájemné vyloučení zprávami

- mailbox *mutex* sdílená n procesy
- $send()$ je asynchronní operací, končí odesláním zprávy
- $receive()$ je synchronní operací, čeká až je mailbox *mutex* neprázdná
- Inicializace: *send(mutex, ´go´);*
- do kritické sekce vstoupí proces P_i který dokončí $receive()$ jako první
- Ostatní procesy budou čekat dokud P_i zprávu „go" nevrátí do schránky

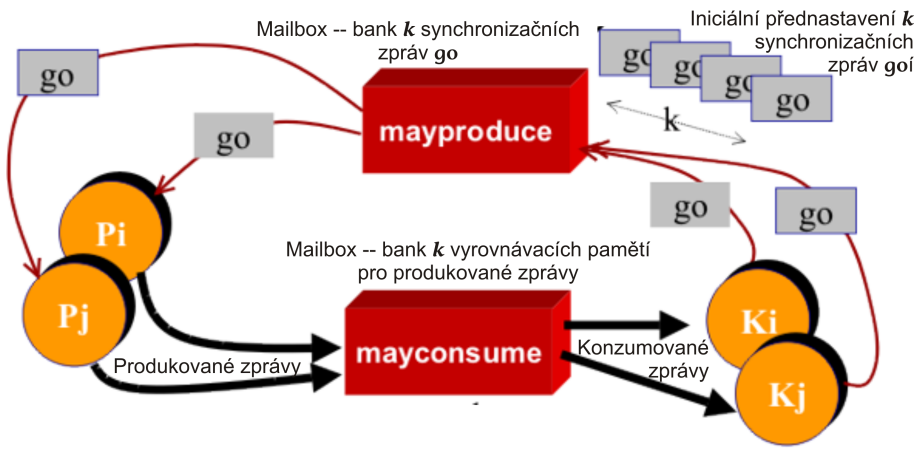
Vzájemné vyloučení zprávami, 2

Process P_i:

var msg: message;
repeat
receive(mutex,msg);
kritická sekce
send(mutex,msg);
zbytek procesu
forever



Producent – konzument zprávami



Producent – konzument zprávami, 2

- Producent umísťuje položky (ve zprávách) do mailboxu / bufferu *mayconsume*
- Konzument může konzumovat položku bufferu obsahující zprávu s daty
- Mailbox *mayproduce* je počátečně vyplněna n prázdnými zprávami (n = rozměr bufferu)
- délka *mayproduce* se produkcí položek zkracuje a konzumací se zvětšuje
- lze podporovat více producentů a konzumentů

Producent – konzument zprávami, 3

Producer:

```
var pmsg: message;
repeat
   receive(mayproduce, pmsg);
   pmsg:= produce();
   send(mayconsume, pmsg);
forever
```

Consumer:

```
var cmsg: message;
repeat
   receive(mayconsume, cmsg);
   consume(cmsg);
   send(mayproduce, go);
forever
```


Čtenáři – písař zprávami zprávami, priorita písaře

- Ke sdíleným údajům přistupuje proces *controller*
- Ostatní procesy ho žádají o povolení vstupu (*writerequest*, *readrequest*), povolení obdrží získáním zprávy *OK*
- Konec přístupu procesy sdělují *controlleru* zprávou *finished*
- V *controlleru* jsou tři *mailboxy*, pro každý typ zprávy jeden
- Proměnná *count* v *controlleru* je inicializována na nejvyšší možný počet čtenářů (např. 100) a platí:
  - ✓ $count > 0$: nečeká žádný písař, mohou být aktivní čtenáři, *controller* může přijmout pouze *finished*
  - ✓ $count = 0$: o přístup žádá pouze písař, *controller* mu pošle *OK* a čeká *finished*
  - ✓ $count < 0$: písař čeká na dokončení aktivních čtenářů, lze přijmout pouze *finished*

Čtenáři – písaři zprávami zprávami

```
void reader(int i)
{
   message rmsg;
      while (true) {
         rmsg = i;
         send (readrequest, rmsg);
         receive (mbox[i], rmsg);
         READUNIT ();
         rmsg = i;
         send (finished, rmsg);
      }
}
void writer(int j)
{
   message rmsg;
   while(true) {
      rmsg = j;
      send (writerequest, rmsg);
      receive (mbox[j], rmsg);
      WRITEUNIT ();
      rmsg = j;
      send (finished, rmsg);
   }
}
```

```
void controller()
{
   while (true)
   {
      if (count > 0) {
         if (!empty (finished)) {
            receive (finished, msg);
            count++;
         }
         else if (!empty (writerequest)) {
            receive (writerequest, msg);
            writer_id = msg.id;
            count = count - 100;
         }
         else if (!empty (readrequest)) {
            receive (readrequest, msg);
            count--;
            send (msg.id, "OK");
         }
      }
      if (count == 0) {
         send (writer_id, "OK");
         receive (finished, msg);
         count = 100;
      }
      while (count < 0) {
         receive (finished, msg);
         count++;
      }
   }
}
```

dokončení už neaktivních čtenářů

potlačení dalších čtenářů

povolí se čtenář

žádný aktivní čtenář, povolí se písař čeká se na jeho konec a nastaví se iniciální stav

čeká se na dokončení aktivních čtenářů

Úloha o 5 večeřících filozofech

- Klasická synchronizační úloha
  - ✓ ilustračně zajímavý problém pro úvodní ilustraci uváznutí

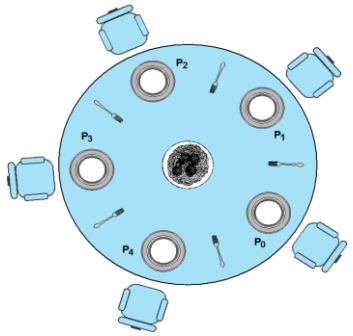
– 5 filozofů buď myslí nebo jí

– jí špagety, ale jen dvěma vidličkami

– co se stane, když se všech 5 filozofů najednou chopí např. levé vidličky ?

– „no přece zemřou hladem, děti"

Hledáme řešení – rituál / protokol – zajišťující ochranu před uváznutím a stárnutím filozofů



Úloha o 5 večeřících filozofech, řešení semafory

```
/* program        diningphilosophers */
semaphore fork [5] = {1};
int   i;
void  philosopher (int i)
{
      while (true) {
            think();
            wait (fork[i]);
            wait (fork [(i+1) mod 5]);
            eat();
            signal(fork [(i+1) mod 5]);
            signal(fork[i]);
      }
}
void main()
{
      parbegin (  philosopher (0), philosopher (1), philosopher (2),
            philosopher (3), philosopher (4));
      }
```


Úloha o 5 večeřících filozofech, ochrana před uváznutím

- zrušení symetrie
  - ✓ jeden filozof je levák, ostatní jsou praváci
  - ✓ levák se liší pořadím získávání vidliček
- Strava se podává n filozofům v jídelně se $n-1$ židlemi
  - ✓ vstup do jídelny hlídá obecný semafor počátečně nastavený na kapacitu $n-1$
  - ✓ řešení chránící jak před uváznutím, tak i před stárnutím

Úloha o 5 večeřících filozofech, ochrana před uváznutím

```
/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
    while (true) {
      think();
      wait (room);
      wait (fork[i]);
      wait (fork [(i+1) mod 5]);
      eat();
      signal (fork [(i+1) mod 5]);
      signal (fork[i]);
      signal (room);
    }

}
void main()
{
    parbegin ( philosopher (0), philosopher (1), philosopher (2),
          philosopher (3), philosopher (4));
}
```


Úloha o 5 večeřících filozofech, ochrana před uváznutím

- Filozof smí uchopit vidličky pouze když jsou obě (jeho levá i pravá) volné
  - ✓ musí je uchopit uvnitř kritické sekce, pro řešení lze použít monitor
  - ✓ definuje se vektor 5 podmínkových proměnných (čekání na vidličku)
  - ✓ definuje se vektor indikující stav vidliček (true = volná)
  - ✓ definují se dvě monitorové procedury pro získání a uvolnění 2 vidliček
  - ✓ uváznutí nehrozí, v monitoru může bý pouze jeden filozof

```
void philosopher[k=0 to 4]          /* the five philosopher clients */
{
   while (true) {
      <think>;
      get_forks(k);          /* client requests two forks via monitor */
      <eat spaghetti>;
      release_forks(k);      /* client releases forks via the monitor */
   }
}
```

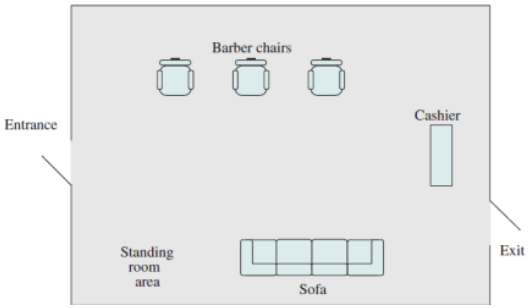

Úloha o 5 večeřících filozofech, řešení monitorem

```
monitor dining_controller;
cond ForkReady[5];         /* condition variable for synchronization */
boolean fork[5] = {true};        /* availability status of each fork */

void get_forks(int pid)          /* pid is the philosopher id number */
{
   int left = pid;
   int right = (++pid) % 5;
   /*grant the left fork*/
   if (!fork(left)
      cwait(ForkReady[left]);         /* queue on condition variable */
   fork(left) = false;
   /*grant the right fork*/
   if (!fork(right)
      cwait(ForkReady(right);         /* queue on condition variable */
   fork(right) = false:
}
void release_forks(int pid)
{
   int left = pid;
   int right = (++pid) % 5;
   /*release the left fork*/
   if (empty(ForkReady[left])      /*no one is waiting for this fork */
      fork(left) = true;
   else                        /* awaken a process waiting on this fork */
      csignal(ForkReady[left]);
   /*release the right fork*/
   if (empty(ForkReady[right])     /*no one is waiting for this fork */
      fork(right) = true;
   else                        /* awaken a process waiting on this fork */
      csignal(ForkReady[right]);
}
```


Komplexní příklad synchronizace, holičství

- V holičství jsou tři holiči, jedna pokladna, čekárna pro 20 zákazníků s pohovkou pro 4 sedící zákazníky, jeden vchod a jeden východ
- Holičství obslouží za den až 50 zákazníků
- Požární předpisy povolují nejvýše 20 zákazníků v provozovně



Komplexní příklad synchronizace, holičství

- Do plné provozovny zákazník nevstupuje
- V čekárně zákazníci podle pořadí příchodu sedí na pohovce, je-li na ní místo, nebo stojí
- Jakmile má holič volno, obsluhuje nejdéle čekajícího zákazníka
- Pohovka se udržuje stále plná, pokud jsou v čekárně alespoň 4 zákazníci
- Ostříhaný zákazník platí holiči u pokladny, holič vybírá peníze od zákazníka. Pokladna je jedna jediná.
- Holič buď stříhá zákazníka nebo přebírá peníze u pokladny nebo spí ve svém křesle – čeká na zákazníka

Holičství, bázové synchronizační (FIFO) semafory

Semaphore	Wait Operation	Signal Operation
max_capacity až 20	Customer waits for space to enter shop.	Exiting customer signals customer waiting to enter.
sofa až 4	Customer waits for seat on sofa.	Customer leaving sofa signals customer waiting for sofa.
barber_chair až 3	Customer waits for empty barber chair. zák. vstane z pohovky až je alsp 1 holič volný	Barber signals when that barber's chair is empty.
cust_ready	Barber waits until a customer is in the chair. zák. si sedl na volné křeslo, budí holiče	Customer signals barber that customer is in the chair.
finished	Customer waits until his haircut is complete. holič sděluje zákazníkovi „hotovo"	Barber signals when cutting hair of this customer is done.
leave_b_chair	Barber waits until customer gets up from the chair. zákazník opustil křeslo	Customer signals barber when customer gets up from chair.
payment	Cashier waits for a customer to pay. pokladník čeká na platbu	Customer signals cashier that he has paid.
receipt	Customer waits for a receipt for payment. je zaplaceno, zák. čeká na stvrzenku	Cashier signals that payment has been accepted.
coord	Wait for a barber resource to be free to perform either the hair cutting or cashiering function.	Signal that a barber resource is free.

Holičství, celkový program

```
/* program barbershop2 */
semaphore max_capacity = 20;
semaphore sofa = 4;
semaphore barber_chair = 3, coord = 3;
semaphore mutex1 = 1, mutex2 = 1;
semaphore cust_ready = 0, leave_b_chair = 0, payment= 0, receipt = 0;
semaphore finished [50] = {0};
int count;

void customer ()                    void barber()          void cashier ()
{                                   {                      {
   ...                                 ...                    ...
}

void main()
{  count := 0;
   parbegin (customer,...50 times,...customer, barber, barber, barber, cashier);
}
```


Holičství, zákazník

```
void customer ()
{
   int custnr;
   semWait(max_capacity);
   enter_shop();
   semWait(mutex1);
   custnr = count;
   count++;
   semSignal(mutex1);
   semWait(sofa);
   sit_on_sofa();
   semWait(barber_chair);
   get_up_from_sofa();
   semSignal(sofa);
   sit_in_barber_chair();
   semWait(mutex2);
   enqueue1(custnr);
   semSignal(cust_ready);
   semSignal(mutex2);
   semWait(finished[custnr]);
   leave_barber_chair();
   semSignal(leave_b_chair);
   pay();
   semSignal(payment);
   semWait(receipt);
   exit_shop();
      semSignal(max_capacity)
}
```

pořadové číslo zákazníka generované po jeho vstupu do holičství
v holičství se může nacházet nejvýše 20 zákazníků

generování pořadového čísla zákazníka, v kritické sekci (*mutex1*)

až 4 nejdéle čekající zákazníci sedí na pohovce (obsluha FIFO),
případní ostatní (až do 20) čekají ve stoje v čekárně

jen tři zákazníci mohou být obsluhování souběžně (obsluha FIFO)
další zákazník může vstát z pohovky (uvolnit na ní místo) a
jít ke křeslům holičů, až bude některý holič volný

zákazník se staví do fronty na volné křeslo a
budí holiče

zákazník je ostříhaný, říká holič

zákazník opouští holičského křesla

zákazník platí

zákazník dostává potvrzení o zaplacení

zákazník odchází

Holičství, holič a pokladna

```
void barber()
{
   int b_cust;
   while (true)
   {
      semWait(cust_ready);
      semWait(mutex2);
      dequeue1(b_cust);
      semSignal(mutex2);
      semWait(coord);
      cut_hair();
      semSignal(coord);
      semSignal(finished[b_cust]);
      semWait(leave_b_chair);
      semSignal(barber_chair);
   }
}
```

čeká na zákazníka

vybírá zákazníka z fronty na volné křeslo

stříhá

sděluje zákazníkovi, že je ostříhaný
čeká až zákazník opustí křeslo
signalizuje volné křeslo

```
Void cashier()
{
   while (true)
   {  semWait(payment);
      semWait(coord);
      accept_pay();
      semSignal(coord);
      semSignal(receipt);
   }
}
```

spouští se platba

vybírá se platba

vydává se potvrzení o zaplacení