

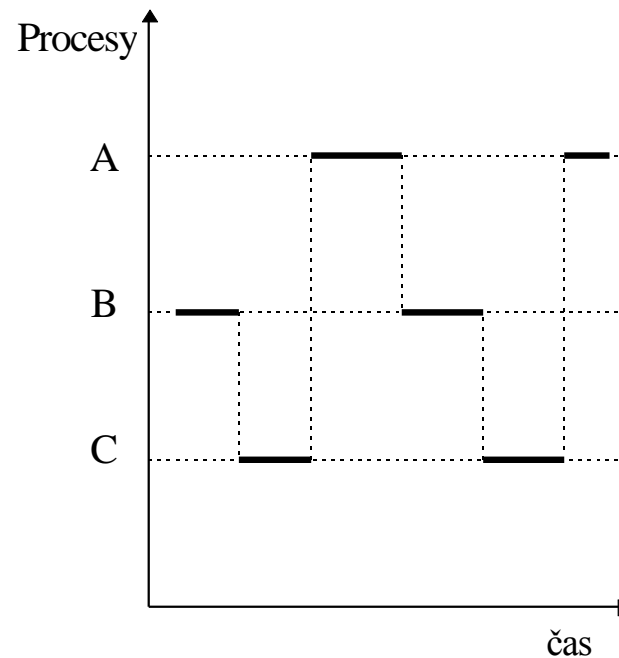
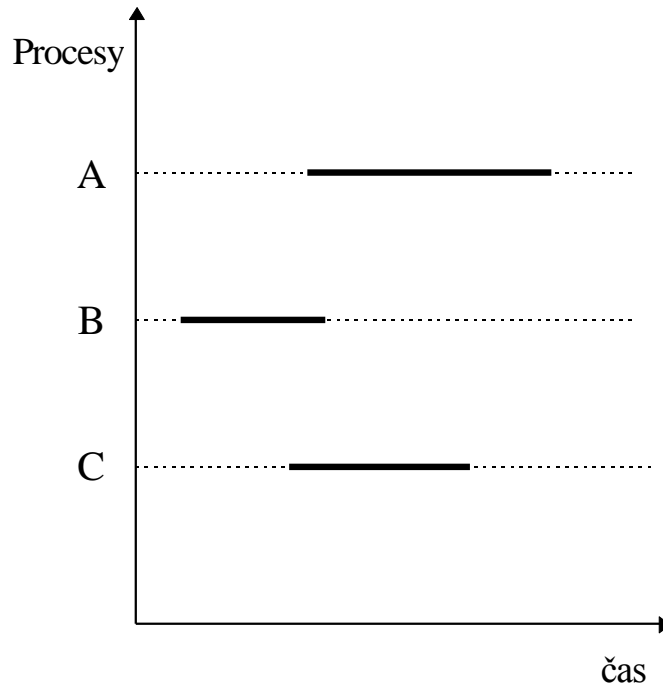
# SYNCHRONIZÁCIA PROCESOV

The slide features a dark brown background. At the bottom, there are two horizontal bars: an orange bar on the left and a blue bar on the right, separated by a thin white line.

# Spôsoby vykonávania procesov

2

- Nezávislé a kooperujúce (spolupracujúce) procesy.
- Paralelné vs. pseudo-paralelné procesy



# Časová závislosť

3

- **Súčasný prístup** k prostriedkom systému môže viesť k **nekonzistencii dát**, neprípustné

## pozorovateľ

úloha: pozorovať (merať )  
a evidovať počet  
pozorovaní.

## *repeat*

*Pozorovanie;* 1.

$C := C + 1;$  2.

*until* koniec;

## reportér

úloha: Má v určitých časových  
úsekoch vypísať počet  
pozorovaní za daný časový  
interval.

## *repeat*

**if** (je čas) **then**

**begin**

writeln(C); 3.

$C := 0;$  4.

**end;**

*until* koniec;

Porovnať výsledky pri poradí vykonania 1,2,3,4 a 1, 3, 2, 4.

# Kedy je potrebná synchronizácia

4

- Ked' synchronizácia je jediným spôsobom zabezpečenia konzistencie dát.
- Ked' vlákna jedného alebo viac procesov využívajú spoločný synchronizačný objekt.
- Ked' synchronizácia môže zabezpečiť bezpečnosť premenlivých dát.
- Pri využívaní súborov mapovaných do pamäte rôznymi vláknami.
- V špeciálnych prípadoch (zarovnanie celých čísel v pamäti).

# Obecné pojmy synchronizácie

5

## □ **Kritická sekcia**

- ▣ sekciu kódu, v ktorej môže používať **zdieľané prostriedky** systému alebo modifikovať zdieľanú informáciu (spoločné premenné, tabuľky, zdieľané súbory a iné)

## □ **Vzájomné vylučovanie**

- protokol, ktorý procesy používajú pri požiadavke vstupu do kritickej sekcie.

*repeat*

*Vstup do KS*

*kritická sekcia*

*Výstup z KS*

*zostávajúca sekcia*

*until false;*

# Obečné pojmy synchronizácie <sup>2</sup>

6

## □ **Atomická** operácia

- operácia, ktorá nemôže byť prerušená, musí sa vykonať len celá.
- napr. operácie čítania a zápisu sú atomické a vzájomné vylúčené.
  - byte read; byte write;
  - word read; word write

# Kritéria správnosti

7

1. **Vzájomné vylúčenie:** ak proces  $P_i$  vykonáva svoju kritickú sekciu, žiadny iný proces nesmie vstúpiť do svojej kritickej sekcie.
2. **Postup:** aspoň jeden z procesov musí dostať povolenie na vstup do kritickej sekcie, ak v nej nie je žiadny iný proces. Výber procesu, ktorý dostane prístup nesmie trvať nekonečne dlho.
3. **Konečné čakanie:** žiadny proces nesmie nekonečne dlho čakať, ak požiadal o vstup do KS

# Princípy pri synchronizácii

8

- Dva základné princípy
  - ▣ Synchronizácia **aktívnym čakaním** - odsun kritickej sekcie sa uskutoční vložení pomocných (obyčajne prázdnych) inštrukcií do kódu procesu.
  - ▣ Synchronizácia **pasívnym čakaním** - odsun kritickej sekcie sa uskutoční dočasným pozastavením procesu, kým sa kritická sekcia neuvolní.



# Prostriedky pre synchronizáciu aktívnym čakaním

9

- **Spoločné premenné** - SW prostriedky, pre dva procesy

## Algoritmus č.1 – **pokus o riešenie**

*repeat*

*while* (*turn*  $\neq$  *i*) *do no-op;*

*kritická sekcia*

*turn* = *j*;

*zostávajúca sekcia*

*until* false;

- **Vzájomné vylúčenie**
  - vyhovuje
- **Postup**
  - nevyhovuje, **musia sa len** striedať
- **Konečné čakanie**
  - vyhovuje

# Spoločné premenné 2

10

## Algoritmus č.2 - **pokus o riešenie**

*var flag: array [0..1] of boolean;*

*repeat*

*flag[i] := true;*  
*while flag[j] do no-op;*

*kritická sekcia*

*flag[i] := false;*

*zostávajúca sekcia*

*until false;*

- **Vzájomné vylúčenie**

- vyhovuje

- **Postup**

- **nevyhovuje**, môže nastať **deadlock** – uviaznutie

- **Konečné čakanie**

- nevyhovuje

# Dekkerove riešenie synchronizácie dvoch procesov spoločnými premennými

11

*repeat*

```
flag[i] := true;
while flag[j] do
  if turn = j then
    begin
      flag[i] := false;
      while turn = j do no-op;
      flag[i] := true;
    end;
```

*kritická sekcia*

```
turn := j;
flag[i] := false;
```

*zostávajúca sekcia*

*until* false;

- **Vzájomné vylúčenie**
  - vyhovuje
- **Postup**
  - vyhovuje
- **Konečné čakanie**
  - vyhovuje

# Synchronizácia $n$ procesov spoločnými premennými

## - algoritmus pekára

12

- Spoločné dátové štruktúry sú:

```
var choosing: array [0.. $n$ -1] of boolean;  
                { inicializované na false }  
    number: array [0.. $n$ -1] of integer;  
                { inicializované na 0 }
```

Pre jednoduchosť definujeme nasledovné pravidlá:

- $(a,b) < (c,d)$ , ak  $a < c$  alebo ak  $a = c$  a  $b < d$ .
- $\max(a_0, \dots, a_{n-1})$  je také číslo  $k$ , pre ktoré platí  $k \geq a_i$  pre  $i = 0, \dots, n-1$ .

# Algoritmus pekára

13

*repeat*

```
choosing[i] := true;  
number[i] := max(number[0], number[1], ..., number[n-1]) + 1;  
choosing[i] := false;  
for j := 1 to n-1 do  
  begin  
    while choosing[j] do no-op;  
    while (number[j] <> 0) and ( (number[j], j) < number[i], i) ) do no-op;  
  end;
```

kritická sekcia

```
number[i] := 0;
```

zostávajúca sekcia

*until false;*

- **Vzájomné vylúčenie** - vyhovuje
- **Postup** - vyhovuje
- **Konečné čakanie** - vyhovuje

# HW prostriedky pre synchronizáciu aktívnym čakaním

14

- **Zákaz prerušenia** - počas modifikácie hodnoty spoločnej premennej
  - **Nedostatky**
    - **nebezpečné pre systém**, pretože musíme spoliehať na jeho korektné použitie, môže byť ohrozený plynulý chod systému.
    - **nehodí pre multiprocessorové systémy**. Zákaz prerušenia pre všetky procesory by spomalil celý systém a ešte by priniesol dodatočné problémy s hodinami, ak sa tieto aktualizujú pomocou prerušení
- **Špeciálne inštrukcie**
  - TSL, SWAP

# Inštrukcia Test-and-Set

15

```
function Test-and-Set ( var target: boolean): boolean;  
begin  
    Test-and-Set := target;  
    target := true ;  
end ;
```

## Použitie

Ak dva procesy potrebujú zaistiť vzájomné vylúčenie v KS, môžu použiť **spoločnú premennú lock**, nastavenú na začiatku na *false*. Nastavenie hodnoty premennej *lock* prebehne vždy **atomicky** a vždy len jeden proces dostane povolenie vstúpiť do KS.

# Použitie TSL

16

*repeat*

*while Test-and-Set(lock) do no-op;*

kritická sekcia

*lock := false ;*

*until false ;*



# Inštrukcia Swap

17

- Zaisťuje **atomickú** výmenu obsahu dvoch premenných

```
procedure Swap ( var a,b: boolean);  
  var temp: boolean;  
  begin  
    temp := a;  
    a := b;  
    b := temp ;  
  end;
```

- Swap môže emulovať TS

```
function test_set(var v: boolean)  
  var t := true;  
  swap (v, t);  
  return t;
```

# Použitie inštrukcie SWAP

18

## Procesy používajú

- jednu spoločnú premennú **lock** (*false*)
- a jednu lokálnu premennú **key**.

*repeat*

```
key := true;  
repeat  
    Swap(lock, key);  
until key = false;
```

kritická sekcia

```
lock := false;
```

zostávajúca sekcia

*until false ;*

# Prehľad doposiaľ popísaných riešení problémov synchronizácie

19

## □ Prehľad

### ■ Vzájomné vylúčenie

- Algoritmus pekára (Bakery Algorithm)
- HW inštrukcie

## □ Nedostatky

### ■ Neriešia všeobecný problém synchronizácie

- len vzájomné vylúčenie

### ■ Používajú aktívne čakanie

- je to plytvanie času procesora

## □ Hľadáme lepšie riešenia

# Synchronizačné problémy

20

- Klasické problémy
  - **Serializácia**
    - $A$  musí predchádzať  $B$
  - **Producent - Konzument**
  - **Čítatelia /Zapisovatelia**
  - **Obedujúci filozofovia**

# Producent - konzument

21

- Dva spolupracujúce procesy
  - ▣ Komunikujú cez vyrovnávaciu pamäť obmedzenej veľkosti.
  - ▣ Prvý proces produkuje informáciu a vkladá ju do vyrovnávacej pamäte,
  - ▣ Druhý proces vyberá z pamäte
- Synchronizačný problém
  - ▣ Aby mohli obidva procesy prebiehať paralelne, ich rýchlosti sa musia zosynchronizovať, t.j. producent musí mať vždy voľné miesto vo vyrovnávacej pamäti pre uloženie informácie a konzument musí mať vždy v pamäti položku, hotovú pre výber.

# Čitateľa /Zapisovateľa

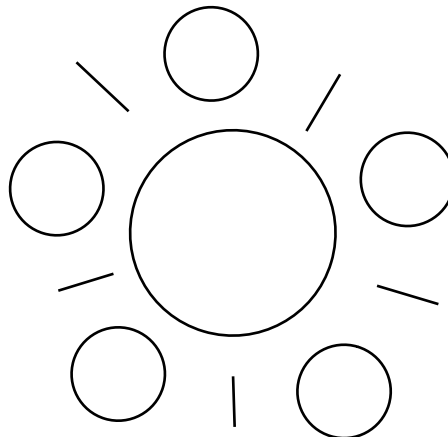
22

- Prístup k zdieľaným dátam X
  - ▣ Dáta môžu **čítať viacerí čitateľa súčasne**
  - ▣ Spolu s čitateľmi nesmie prístup k dátam žiadny zapisovateľ
  - ▣ **Zapisovateľa** prístupujú k dátam **vždy po jednom!**
  - ▣ Tento spôsob prístupov zaistí konzistenciu zdieľaných dát
  - ▣ Vzájomné vylúčenie je príliš obmedzujúce

# Obedujúci filozofovia

23

- Popis synchronizačného problému
  - ▣ Je päť filozofov, ktorí buď jedia alebo premýšľajú a na stole je 5 tanierov a 5 vidličiek medzi nimi
  - ▣ Ak sa chce filozof najesť potrebuje **dve** vidličky
  - ▣ V danom okamihu filozof **môže zobrať len jednu**
- Synchronizačný problém
  - nedovoliť filozofovi držať jednu vidličku a čakať na uvoľnenie druhej, pretože môže nastať uviaznutie.



# Zle riešenie

24

## □ Algoritmus filozofa

- Rozhodne sa jest'
- Čaká kým sa uvoľní ľavá vidlička
- Zoberie ju
- Čaká kým sa uvoľní pravá vidlička
- Zoberie ju
- Je
- Uvoľní ľavú vidličku
- Uvoľní pravú vidličku
- Rozmýšľa

## □ Prečo je to riešenie zlé?

- Môže vzniknúť **uviaznutie** (všetci budú hladovať)

## □ Iné algoritmy

- ako predchádzajúce, ale neberie ľavú vidličku, ak práva nie je voľná
- asymetrický postup párnych a nepárnych filozofov pri získaní vidličiek
- variant algoritmu pekára
- Pozor – neexistencia uviaznutia neodstraňuje starváciu!





# Semaforey

25

- Celočíselná premenná  $S$
- Pristupuje sa k nej pomocou dvoch atomických operácií
- WAIT (niekedy označovaná ako  $P$ )
  - Pokiaľ  $S \leq 0$  do čakaj;
  - $S := S - 1$ ;
- SIGNAL (niekedy označovaná ako  $V$ )
  - $S := S + 1$ ;

*wait(S):* while  $S \leq 0$  do no-op;  
                   $S := S - 1$ ;

*signal(S):*  $S := S + 1$ ;

# Atomické operácie

26

## ▣ Presnejšie atomické operácie sú

- operácia test-and-decrement

```
boolean function Ok();  
begin  
    if  $S \leq 0$  then begin  
         $S := S - 1$ ;  
        return false; end  
    else return true;  
end;
```

- operácia increment

```
 $S := S + 1$ ;
```

## ▣ Potom operácia **P** je **blokujúca**

## ▣ Operácia **V** **nie je blokujúca**

# Použitie semaforov

27

## □ **Vzájomné vylúčenie procesov v kritickej sekcii**

- semafor sa inicializuje na 1

- $P(\text{Sem});$  KS;  $V(\text{Sem});$

## □ **Serializácia – zachovanie poradia vykonávania procesov**

- semafor sa inicializuje na 0

- proces A

- $P(\text{Sem});$  A( );

- proces B

- B( );  $V(\text{Sem});$

## □ **Počítanie**

- semafor počet sa inicializuje na N

- $P(\text{počet});$  odstráni položku //zníži počet;

- vloží položku;  $V(\text{počet});$  //zvýši počet

## □ **Príklad – Producent – Komzument**

# Čitatelia/Zapisovatelia pomocou semaforov

28

## □ Použité semaforey

- mutex (pre vzájomnú vylúčenie)

- zapisovatelia

  - Čitatelia sa synchronizujú so zapisovateľmi pomocou premennej *writer*

  - A medzi sebou pri modifikácií premennej *readcount* sa synchronizujú pomocou mutex-u

- zapisovatelia

  - `P(writer);`

  - `WRITE.....`

  - `V(writer);`

# Úloha čitateľa - zapisovateľa pomocou semaforov

29

```
var mutex, {zaist'uje vzájomné vylúčenie pri zmene hodnoty premennej readcount}  
wrt: semaphore; {zaist'uje povolenie čítania}  
readcount: integer; {počet procesov, ktorí čítajú}
```

## Proces čitateľ:

```
wait(mutex) ;  
    readcount := readcount + 1 ;  
  
    if readcount = 1 then wait(wrt) ;  
signal(mutex) ;  
    ...  
    vykoná sa čítanie  
    ...  
wait(mutex) ;  
    readcount := readcount - 1 ;  
    if readcount = 0 then  
        signal(wrt) ;  
signal(mutex) ;
```

## Proces zapisovateľ:

```
wait(wrt) ;  
    ...  
    vykoná sa zápis  
    ...  
signal(wrt) ;
```

# Úloha producent - konzument pomocou semaforov

30

## Použité semaforey:

- mutex*: semafor; { inicializovaný na 0, zaist'uje vzájomné  
vylúčenie procesov pri práci s bufrom }
- empty*: semafor; { počíta prázdne položky v bufri inicializovaný  
na  $N$  }
- full* : semafor; { počíta plné položky, inicializovaný na 0 }

# Úloha producent - konzument pomocou semaforov 2

31

## Proces producent:

```
repeat
    ...
    vyrobí položku do nextp
    ...
    wait(empty) ;
    wait(mutex) ;
    ...
    vloží nextp do bufra
    ...
    signal(mutex) ;
    signal(fill) ;
until false;
```

## Proces konzument:

```
repeat
    wait(fill) ;
    wait(mutex) ;
    ...
    vyberie jednu položku z bufra do nextc
    ...
    signal(mutex) ;
    signal(empty) ;
    ...
    spracuje položku z nextc
    ...
until false;
```

# Binárne semaforey

32

- Semaforey popísané v predchádzajúcej časti sú známe ako **počítajúce (counting) semaforey**, pretože ich hodnota sa môže meniť neobmedzene.
- **Binárny semafor** — zabezpečuje len vzájomné vylúčenie
  - nadobúda hodnoty len 0 alebo 1
  - ľahšie sa implementuje
  - je univerzálny, lebo s jeho pomocou sa dajú implementovať aj počítajúce semaforey



# Implementácia počítajúceho semafora pomocou binárnych semaforov <sup>1</sup>

33

- Potrebujeme tieto štruktúry:

binárne semaforey: S1, S2

int C

**S1 = 1**

{ vzájomné vylúčenie pri zmene hodnoty semafora }

**S2 = 0**

{ blokovanie procesu, kým sa hodnota semafora zvýši }

**C=N**

{ hodnota semafora }

# Implementácia počítajúceho semafora pomocou binárnych semaforov <sup>2</sup>

34

Operácia wait:

```
wait(S1);  
C--;  
if (C < 0) {  
    signal(S1);  
    wait(S2);  
}  
signal(S1);
```

Operácia *signal*

```
wait(S1);  
C ++;  
if (C <= 0)  
    signal(S2);  
else  
    signal(S1);
```

# Ukážka univerzálnosti

35

## □ Operácia *wait*

```
wait(S1);  
C--;  
if (C < 0) {  
    signal(S1);  
    wait(S2);  
}  
signal(S1);
```

## □ Operácia *signal*

```
wait(S1);  
C ++;  
if (C <= 0)  
    signal(S2);  
else  
    signal(S1);
```

# Uviaznutie a starvacia pri použití semaforov

36

## □ Príklad uviaznutia

- Pre ilustráciu tejto situácie - systém s dvomi procesmi -  $P_0$  a  $P_1$ , ktoré používajú dva semaforey  $S$  a  $Q$ , nastavené na hodnotu 1:

$P_0$	$P_1$
$wait(S);$	$wait(Q);$
$wait(Q);$	$wait(S);$
.	.
.	.
.	.
$signal(S);$	$signal(Q);$
$signal(Q);$	$signal(S);$

- Príklad starvacie
  - ▣ pri čitateľoch/zapisovateľoch
  - ▣ ak je nekonečný prúd čitateľov *nikdy nedovolí* zapisovateľovi vstup do KS a naopak
  - ▣ Semaforey negarantujú postup

# Nedostatky semaforov s aktívnym čakaním

38

- Hlavný nedostatok synchronizácie pomocou semaforov - vyžadujú aktívne čakanie procesu v prípade, že sa proces nemôže dostať do KS (*spinlock*)
- V multiprogramových systémoch - vážny nedostatok, pretože proces, ktorý nerobí nič, dostáva pridelený čas procesora
- Použijeme štruktúru s nasledovnou sémantikou

*type semaphore = record*

*value: integer;*      {hodnota semafora}


*L: list of process;*    {zoznam procesov}


*end;*

- Front na semafore - obyčajne FIFO, aby sa zaistilo poradie v ktorom procesy žiadali o prostriedok

# Modifikácie operácií *wait* a *signal* pre zaistenie pasívneho čakania

39

```
wait(S):  S.value := S.value - 1;  
          if S.value < 0 then  
            begin  
              pridaj proces do S.L;  
              zablokuj volajúci proces ;   
            end;
```

```
signal(S):  S.value := S.value + 1;  
            if S.value ≤ 0 then  
              begin  
                odstráň proces P z S.L;  
                odblokuj proces P;   
              end;
```

# Chyby pri použití semaforov

40

- Semaforey sú náchylné na chyby!
- Príklady
  - Vynechanie *wait* a/alebo *signal*
    - nenastane vzájomné vylúčenie
  - Zámená poradia *wait* a *signal*
    - **uviaznutie!**
  - operácia *wait*, nasledovaná operáciou *wait*



# Podpora v programovacích jazykoch

41

- Programovacie jazyky podporujúce paralelné procesy
- Príklady
  - Communicating Sequential Procedures (Hoare)
  - Concurrent Pascal
  - Modula 2
  - Ada83, Ada95
  - Novšie objektovo-orientované jazyky
    - Concurrent C, Java, C++, C#

# Monitory<sub>1</sub>

42

- Je to skupina, pozostávajúca z
  - vlastných dát,
  - verejných procedúr,
  - podmienkových premenných (condition variables) (fronty procesov).
- Každá procedúra môže mať
  - ▣ Lokálne premenné
  - ▣ Formálne parametre
- **Vykonanie procedúr je vzájomné vylúčené**

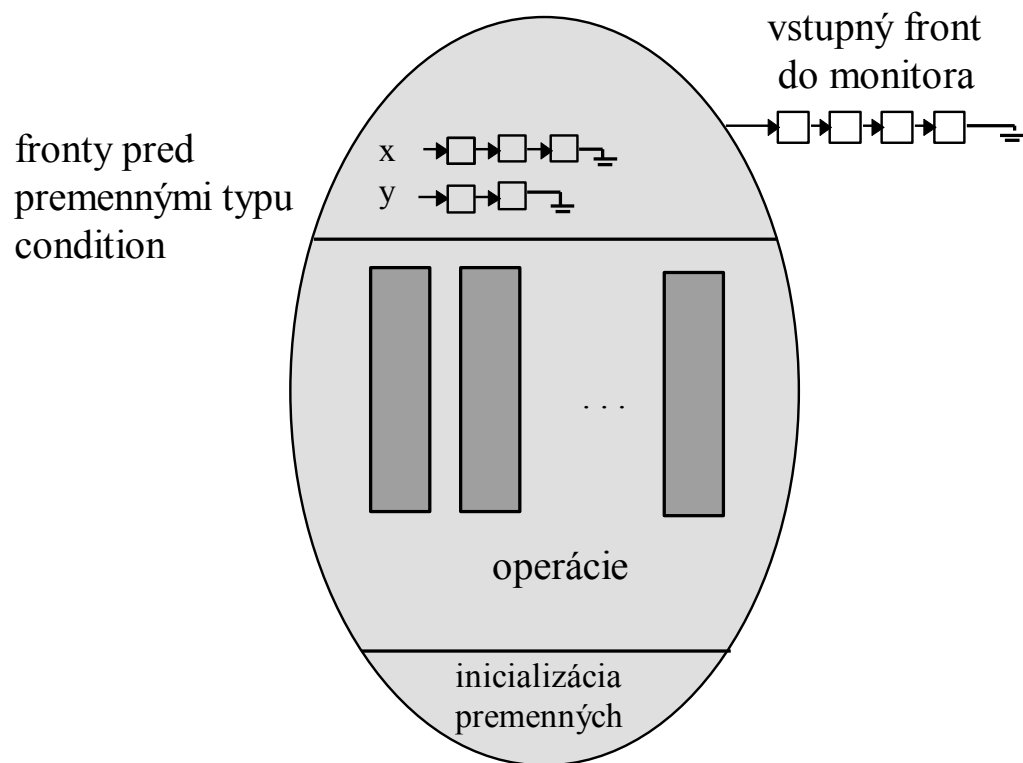
# Monitory a podmienkové premenné

43

- ▣ Monitory poskytujú dodatočný dátový typ, ktorý dovoľuje zablokovat' proces v prípade, že nie je splnená určitá podmienka.
- ▣ **typ *condition***
- ▣ jediné operácie nad ním sú *wait* a *signal*.
- ▣ Operácia ***wait(condition)***:
  - uvoľní uzamknutie monitora a „uspí“ proces. Keď sa proces „zobudí“, získa znova prístup k monitoru.
- ▣ Operácia ***signal(condition)***:
  - „zobudí“ jeden z procesov čakajúcich na premennú typu *condition*. Ak vo fronte nie je žiadny proces, neurobí nič.

# Monitor

44



- ▣ Sémantika **dovoľuje niekoľkým procesom čakať „vo vnútri“ monitora**, pričom sa vykonáva len jeden.
- ▣ Čakacie fronty sú vnútorné pre monitor, *wait* a *signal* su volané znútra.
- ▣ Niekoľko rôznych variantov **chovania procesov po použití operácií *signal***

Nech **proces *P* zavolať operáciu *signal*** nad premennou typu *condition*, **pred ktorou je pozastavený proces *Q***.

Sú dve možnosti, ktoré vylučujú prítomnosť obidvoch procesov v monitore:

- **Proces *P* buď čaká, kým *Q* opustí monitor,**  
alebo čaká na inú podmienku.
- **Proces *Q* čaká, pokiaľ *P* opustí monitor,**  
alebo čaká na inú podmienku

# Problémy

46

- Monitory musia mať dobrý algoritmus plánovania, aby nedochádzalo k starvácií
  - ▣ na úrovni prístupu k monitoru,
  - ▣ pri „zobudení“ procesov, čakajúcich na podmienku.

# Úloha obedujúcich filozofov pomocou monitora 1

47

```
type dining-philosophers = monitor
```

```
var state: array[0..4] of (thinking, hungry, eating);
```

```
    self: array [ 0..4] of condition;
```

```
procedure entry pickup(i: 0..4 );
```

```
{ked' má hlad, nastaví svoj stav na hladný, otestuje  
susedov, či mu nedržia vidličky, a ak tomu tak nie je, je, ináč čaká}
```

```
begin
```

```
    state[i] := hungry;
```

```
    test(i);
```

```
    if state[i] <> eating then self[i].wait;
```

```
end
```

# Úloha obedujúcich filozofov pomocou monitora <sup>2</sup>

48

```
procedure entry putdown(i: 0..4) ;  
  { po ukončení jedla nastavím svoj stav na myslenie a oznámim to susedom}  
  begin  
    state[i] := thinking;  
    test((i+4) mod 5 );  
    test((i+1) mod 5 );  
  end;  
  
procedure test(k: 0..4);  
begin  
  if (state[k+4 mod 5] <> eating) and (state[k]=hungry)  
    and (state[k+1 mod 5]<>eating) then  
    begin  
      state[k]:=eating;  
      self[k].signal;  
    end;  
end;
```



# Úloha obedujúcich filozofov pomocou monitora <sup>3</sup>

49

**begin**

*{inicializácia stavu filozofov - rozmýšľanie}*

**for**  $i:=0$  **to** 4 **do**

$state[i] := thinking;$

**end;**

# Synchronizácia vlákien v knižnici *pthread* ---

50

- Interfejs knižnice pthread je nezávislý na OS
- Poskytuje:
  - ▣ mutex
  - ▣ premenné typu condition (podmienkové)
  - ▣ semaforey

A neportabilné (závislé od OS) synchronizačné prostriedky:

- ▣ read-write locks
- ▣ spin locks

# Synchronizácia vlákien v knižnici *pthread*

51

## Mutexy

- ▣ `pthread_mutex_t mutex;`
- ▣ `const pthread_mutexattr_t attr;`
- ▣ `int status;`
  
- ▣ `status = pthread_mutex_init(&mutex,&attr);`
- ▣ `status = pthread_mutex_destroy(&mutex);`
- ▣ `status = pthread_mutex_lock(&mutex);` - *zablokuje*
- ▣ `status = pthread_mutex_unlock(&mutex);` - *odblokuje*
- ▣ `status = pthread_mutex_trylock(&mutex);`

# Synchronizácia vlákien v knižnici *pthread*

52

## Premenné typu **condition** (podmienkové)

- `int status;`
- `pthread_condition_t cond;`
- `const pthread_condattr_t attr;`
- `pthread_mutex mutex;`
- `status = pthread_cond_init(&cond,&attr);`
- `status = pthread_cond_destroy(&cond);`
- `status = pthread_cond_signal(&cond);`
- `status = pthread_cond_broadcast(&cond);`

# Synchronizácia vlákien v knižnici *pthread*

53

## Semafor

- `int status, pshared;`
- `sem_t sem;`
- `unsigned int initial_value;`
  
- `status = sem_init(&sem, pshared, initial_value);`
- `status = sem_destroy(&sem);`
- `status = sem_wait(&sem);`    */\* wait - zablokuje \*/*
- `status = sem_post(&sem);`    */\* signal - odblokuje \*/*
- `status = sem_trywait(&sem);`

# Synchronizácia v niektorých OS

54

## □ Solaris 2 - multi-CPU OS

### ▣ len pre krátkodobý prístup

- adaptívne mutexy
- štandardné *spinlock* semaforey
- ak lock je použitý v bežiacom vlákne, aktívne čakanie - spin
  - ináč sa zablokuje

### ▣ pre zložitejšie situácie

- Podmienkové premenné
  - wait a signal
- Reader-writer locks
  - pre častý, väčšinu read-only prístup

# Synchronizácia v Linuxe

55

- Linux:
  - ▣ Pred v. 2.6, zákaz prerušení pre implementáciu krátkych kritických sekcií
  - ▣ Od v.2.6 plne preemptívne jadro
- Pre synchronizáciu Linux poskytuje:
  - ▣ Semaforey
  - ▣ Atomické celé čísla
  - ▣ Spinlock-y
  - ▣ reader-writer zámky
- V jednoprocessorových systémoch spinlock-y sú nahradené umožnením/znemožnením preempcie jadra