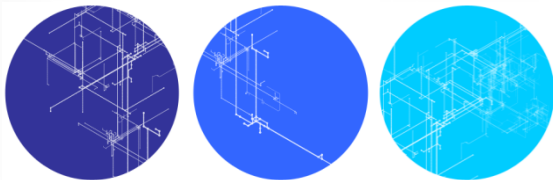


# PB153 OPERAČNÍ SYSTÉMY A JEJICH ROZHRAŇÍ



**Synchronizace procesů**

**12**

# PARALELNÍ BĚH PROCESŮ

- Synchronizace běhu procesů
  - jeden proces čeká na událost z druhého procesu
- Komunikace mezi procesy
  - komunikace – způsob synchronizace, koordinace různých aktivit
  - může dojít k uváznutí
    - každý proces čeká na zprávu od nějakého jiného procesu
  - může dojít ke stárnutí
    - dva procesy si opakovaně posílají zprávy zatímco třetí proces čeká na zprávu nekonečně dlouho
- Sdílení prostředků
  - procesy používají a modifikují sdílená data
  - operace zápisu musí být vzájemně výlučné
  - operace zápisu musí být vzájemně výlučné s operacemi čtení
    - operace čtení mohou být realizovány souběžně
  - pro zabezpečení integrity dat se používají kritické sekce

# NEKONZISTENCE

- Paralelní přístup ke sdíleným údajům může být příčinou nekonzistence dat
- Udržování konzistence dat vyžaduje používání mechanismů, které zajistí patřičné provádění spolupracujících procesů
- Problém komunikace procesů v úloze typu Producent-Konzument přes vyrovnávací paměť s omezenou kapacitou

# PRODUCENT-KONZUMENT (1)

- Sdílená data

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

# PRODUCENT-KONZUMENT (2)

- Producent

```
item nextProduced;
```

```
while (1) {
```

```
    while (counter == BUFFER_SIZE)
```

```
        ; /* do nothing */
```

```
    buffer[in] = nextProduced;
```

```
    in = (in + 1) % BUFFER_SIZE;
```

```
    counter++;
```

```
}
```

# PRODUCENT-KONZUMENT (3)

- Konzument

```
item nextConsumed;
```

```
while (1) {
```

```
    while (counter == 0)
```

```
        ; /* do nothing */
```

```
    nextConsumed = buffer[out];
```

```
    out = (out + 1) % BUFFER_SIZE;
```

```
    counter--;
```

```
}
```

# PRODUCENT-KONZUMENT (4)

- Atomická operace je taková operace, která vždy proběhne bez přerušení
- Následující příkazy musí být atomické
  - **counter++;**
  - **counter--;**
- **count++** v assembleru může vypadat
  - **register1 = counter**
  - **register1 = register1 + 1**
  - **counter = register1**
- **count--** v assembleru může vypadat
  - **register2 = counter**
  - **register2 = register2 - 1**
  - **counter = register2**

# PRODUCENT-KONZUMENT (5)

- Protože takto implementované operace `count++` a `count--` nejsou atomické, můžeme se dostat do problémů s konzistencí
- Necht' je hodnota **counter** nastavena na 5. Může nastat:
  - producer: **register1 = counter** (*register1 = 5*)
  - producer: **register1 = register1 + 1** (*register1 = 6*)
  - consumer: **register2 = counter** (*register2 = 5*)
  - consumer: **register2 = register2 - 1** (*register2 = 4*)
  - producer: **counter = register1** (*counter = 6*)
  - consumer: **counter = register2** (*counter = 4*)
- Výsledná hodnota proměnné **counter** bude buďto 4 nebo 6, zatímco správný výsledek má být 5.



# ANIMACE: PRODUCENT-KONZUMENT (5)



# RACE CONDITION

- Race condition (podmínka soupeření):
  - více procesů současně přistupuje ke sdíleným zdrojům a manipulují s nimi
  - konečnou hodnotu zdroje určuje poslední z procesů, který zdroj po manipulaci opustí
- Ochrana procesů před negativními dopady race condition
  - je potřeba procesy synchronizovat

# PROBLÉM KRITICKÉ SEKCE

- N procesů soupeří o právo používat jistá sdílená data
- V každém procesu se nachází segment kódu programu nazývaný *kritická sekce*, ve kterém proces přistupuje ke sdíleným zdrojům
- Problém:
  - je potřeba zajistit, že v kritické sekci, sdružené s jistým zdrojem, se bude nacházet nejvýše jeden proces

# PODMÍNKY ŘEŠENÍ PROBLÉMU KRITICKÉ SEKCE

- Podmínka vzájemného vyloučení (mutual exclusion), podmínka bezpečnosti, „safety“
  - jestliže proces P1 provádí svoji kritickou sekci, žádný jiný proces nemůže provádět svoji kritickou sekci sdruženou se stejným zdrojem
- Podmínka trvalosti postupu (progress), podmínka živosti, „liveliness“
  - jestliže žádný proces neprovádí svoji sekci sdruženou s jistým zdrojem a existuje alespoň jeden proces, který si přeje vstoupit do kritické sekce sdružené s tímto zdroje, pak výběr procesu, který do takové kritické sekce vstoupí, se nesmí odkládat nekonečně dlouho
- Podmínka konečnosti doby čekání (bounded waiting), podmínka spravedlivosti, „fairness“
  - musí existovat horní mez počtu, kolikrát může být povolen vstup do kritické sekce sdružené s jistým zdrojem jiným procesům než procesu, který vydal žádost o vstup do kritické sekce sdružené s tímto zdrojem, po vydání takové žádosti a před tím, než je takový požadavek uspokojen
  - předpokládáme, že každý proces běží nenulovou rychlostí
  - o relativní rychlosti procesů nic nevíme

# POČÁTEČNÍ NÁVRHY ŘEŠENÍ

- Máme pouze 2 procesy,  $P_0$  a  $P_1$
- Generická struktura procesu  $P_i$

do {

entry section

critical section

exit section

reminder section

} while (1);

- Procesy mohou za účelem dosažení synchronizace svých akcí sdílet společné proměnné
- Činné čekání na splnění podmínky v „entry section“ – „busy waiting“

# ŘEŠENÍ PROBLÉMU KS

- Softwarová řešení
  - algoritmy, jejichž správnost se nespolehá na žádné další předpoklady
  - s aktivním čekáním „busy waiting“
- Hardwarová řešení
  - vyžadují speciální instrukce procesoru
  - s aktivním čekáním
- Řešení zprostředkované operačním systémem
  - potřebné funkce a datové struktury poskytuje OS
  - s pasivním čekáním
  - podpora v programovacím systému/jazyku
    - semafore, monitory, zasílání zpráv

# SITUACE BEZ PODPORY OS

- Nedostatek softwarového řešení
  - procesy, které žádají o vstup do svých KS to dělají metodou „busy waiting“
    - po nezanedbatelnou dobu spotřebovávají čas procesoru
- Speciální instrukce
  - výhody
    - vhodné i pro multiprocesory (na rozdíl od prostého maskování/povolení přerušení)
  - nevýhody
    - opět „busy waiting“
    - možnost stárnutí – náhodnost řešení konfliktu
    - možnost uváznutí v prioritním prostředí (proces v KS nedostává čas CPU)

# SEMAFORY

- Synchronizační nástroj, který lze implementovat i bez „busy waiting“
  - proces je (operačním systémem) „uspán“ a „probuzen“
  - tj. řešení na úrovni OS

- Definice

Semaphore  $S$  : integer

- Lze ho zpřístupnit pouze pomocí dvou atomických operací

**wait (S):**

```
while  $S \leq 0$  do no-op;  
S --;
```

**signal(S):**

```
S ++;
```



# KRITICKÁ SEKCE

- Sdílená data:

```
semaphore mutex; // počátečně mutex = 1
```

- Proces  $P_i$ :

```
do {
```

```
    wait(mutex);
```

```
        critical section
```

```
    signal(mutex);
```

```
        remainder section
```

```
} while (1);
```

# SYNCHRONIZACE SEMAFOREM

- Má se provést akce B v  $P_j$  pouze po té, co se provede akce A v  $P_i$
- Použije se semafor flag inicializovaný na 0
- Program:

$P_i$

⋮

A

*signal(flag)*

$P_j$

⋮

*wait(flag)*

B

# UVÁZNUTÍ A STÁRNUTÍ

- Uváznutí
  - dva nebo více procesů neomezeně dlouho čekají na událost, kterou může generovat pouze jeden z čekajících procesů
  - Necht' S a Q jsou dva semaforey inicializované na 1

$P_0$	$P_1$
<i>wait(S);</i>	<i>wait(Q);</i>
<i>wait(Q);</i>	<i>wait(S);</i>
⋮	⋮
<i>signal(S);</i>	<i>signal(Q);</i>
<i>signal(Q)</i>	<i>signal(S);</i>

- Stárnutí
  - neomezené blokování, proces nemusí být odstraněn z fronty na semafor nikdy (předbíhání vyššími prioritami, ...)

# DVA TYPY SEMAFORŮ

- Obecný semafor  $S$ 
  - celočíselná hodnota z neomezovaného intervalu
- Binární semafor
  - celočíselná hodnota z intervalu  $\langle 0, 1 \rangle$
- Implementovatelnost
  - binární semafor lze snadněji implementovat
  - obecný semafor lze implementovat semaforem binárním

Výukovou pomůcku zpracovalo  
**Servisní středisko pro e-learning na MU**

CZ.1.07/2.2.00/28.0041

Centrum interaktivních a multimediálních studijních opor pro inovaci výuky a efektivní učení



INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ