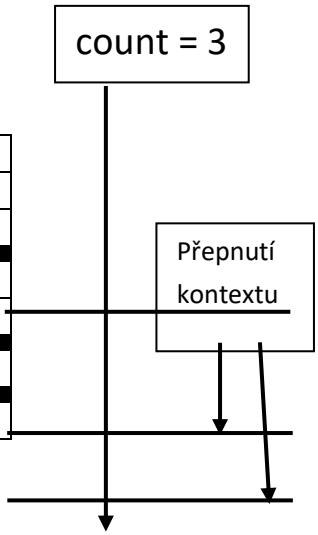
SYNCHRONIZACE PROCESŮ

- **Problém Producent vs. Konzument**
- **Kritická sekce**
  - **Charakteristika**
  - **Podmínky kritické sekce, ošetření**
  - **Aktivní a pasivní čekání**
- **Sdílený prostředek**
- **Řešení kritické sekce**
  - **Zákaz přerušení**
  - **Zamykací proměnná**
  - **Přesné střídání**
  - **Petersonovo řešení**
  - **Atomická instrukce**
  - **Sleep() a Wakeup()**
  - **Semafory a transakce**
- **Klasické synchronizační problémy**

Producent vs. Konzument

- 1 z nich bude počitadlo (counter) -> počítá hodnoty v bufferu
- 2 z nich bude buffer
- Producent přidá hodnoty do bufferu a konzument je vezme

Běží	Akce	Výsledek
Producent	R0 = count	R0 = 3
Producent	R0++	R0 = 4
Konzument	R1 = count	R1 = 3
Konzument	R1 --	R1 = 2
Producent	count = R0	count = 4
Konzument	count = R1	count = 2

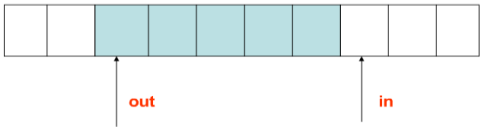


- Po vykonání instrukce se může přepnout kontext
- Producent generuje data do vyrovnávací paměti (buffer) o velikostí n (společná paměť pro producenta a konzumenta)
- Konzument z bufferu odebírá
- Po vygenerování nových dat producentem se zvyšuje hodnota počitadla (count ++)
- Konzument po odebrání dat z bufferu sníží hodnotu count --
- Bez správného ošetření přístupu k bufferu a proměnné count není zaručen správný výsledek a může hrozit uváznutí procesu
- V jazycích vyšší úrovně je inkrementace otázkou jediného příkazu, ve skutečnosti je tato operace otázkou více instrukcí (assembler) a k přepnutí kontextu dochází až po provedení instrukce
- Je nutná komunikace mezi procesy například pomoci IPS

- Producent
  - Generuje data do vyrovnávací paměti s konečnou kapacitou (buffer)
- Konzument
  - Z této paměti data odebírá
- Buffer
  - Společná paměť slouží k výměně dat

Synchronizace procesů

- Program producent a konzument
- výměna dat ve společné paměti **BUFFER**
- producent vkládá data na adrese **in**
- konzument vybírá data na adrese **out**



n=10 … velikost bufferu
pocitadlo … společná proměnná

```
repeat
  … produkuj data v ….    produkuj
while pocitadlo = n do nic
   buffer [in] := produkuj
   in := in + 1 mod n
   pocitadlo := pocitadlo + 1
until false
```

```
repeat
  … konzumuj data v ….   konzumuj
while pocitadlo = 0 do nic
    konzumuj  := buffer [out]
    out := out + 1 mod n
    pocitadlo := pocitadlo - 1
until false
```

binární kód

```
register1 := pocitadlo
register1 := register1 + 1
pocitadlo := register1
```

```
register2 := pocitadlo
register2 := register2 -1
pocitadlo := register2
```

- **producent:**
  - pokud proměnná počítadlo má stejnou velikost jako velikost bufferu, nemůže nic produkovat, jinak vyprodukuje data, vloží do společné paměti a inkrementuje počítadlo (aktualizuje počet prvků v bufferu)

- **konzument:**
  - pokud proměnná počítadlo je 0, tudíž v bufferu nejsou žádná data, nemůže také konzument nic z bufferu vybírat, pokud data v bufferu jsou, vybere data a sníží proměnnou počítadlo o 1

Kritická sekce

- část strojového kódu, kde dochází k přístupu ke sdíleným prostředkům
- hrozí zde přístup více vláken najednou
- **Podmínky KS:**
    - Žádné dva procesy nebo vlákna nesmí být současně uvnitř stejné kritické sekce
    - Na řešení nesmí mít vliv počet CPU
    - Žádný proces nesmí zůstat čekat nekonečně dlouho na kritickou sekci
    - Žádný proces mimo kritickou sekci nesmí blokovat jiný proces, který by do něj měl vstoupit
- **Řešení KS:**
    - Softwarové řešení na aplikační úrovni
        - Je v celé režii programátora, jedno ze základních řešení, je zde aktivní čekání
    - Hardwarové řešení
        - Pomocí speciálních instrukcí procesů a i zde je aktivní čekání
    - Softwarové řešení zprostředkované OS
        - Stára se o to jádro OS, je zde pasivní čekání
    - Aktivní čekání
        - testuje proměnou, jestli může vstoupit do kritické sekce
    - Pasivní čekání
        - uspí se a proces, který chce vstoupit do kritické sekce
        -

Řešení kritické sekce

- vzájemné vyloučení s aktivním čekání
    - **zákaz přerušení**
        - Nejjednodušší řešeni
        - Zákaz všech přerušení procesy, které právě vstoupily do kritické sekce
        - Opětovné povolení přerušení po opuštění kritické sekce
        - Pokud není povoleno přerušení, nedojde k přepnutí na jiný proces
        - Nevhodné řešení, protože uživatelský proces zasahuje do chodu systému tím, že zakazuje přerušení
        - Pokud opětovně nepovolil přerušení, tak systém spadne
        - Pokud je systém více procesorový, tak proces běžící na jiném procesoru může vstoupit do kritické sekce

- **Zamykací proměnná**
  - Každý proces dostane zamykací proměnnou lock, když je v 1 nemůže do kritické sekce, 0 můžu, je tam volno
  - Může nastat situace, že jeden proces ji nestihne načíst před přepnutím kontextu, druhý proces začne taky načítat lock proměnnou -> už jsou dva procesy v kritické sekci

Zamykací proměnné

– Kritickou sekci „ochráníme" sdílenou zamykací proměnnou přidruženou ke sdílenému prostředku (iniciálně = 0).
  - proměnná **lock**

– Před vstupem do kritické sekce proces testuje tuto proměnnou a, je-li nulová, nastaví ji na 1 a vstoupí do kritické sekce. Neměla-li proměnná hodnotu 0, proces čeká ve smyčce (aktivní čekání – *busy waiting*).

```
while(lock != 0)
        ; /* Nedělej nic a čekej */
```

– Při opouštění kritické sekce proces tuto proměnnou opět nuluje

```
lock = 0;
```

– Nevyřešili jsme však nic: souběh jsme přenesli na zamykací proměnnou
  - Myšlenka zamykacích proměnných však není zcela chybná

- **Přesné střídání**
  - Vylepšení předchozího řešení (řešení č. 2)
  - Proměna turn určuje, který proces může vstoupit do kritické sekce
  - Proces otestoval turn, že je rovné 0, takže vstoupí do kritické sekce
  - Když P0 opustí kritickou sekci, nastaví turn na 1, což říká že P1 může vstoupit
  - Přijde P1, uvidí, že turn = 1, což znamená, že může vstoupit do kritické sekce, po vystoupení nastaví turn na 0
  - P0 blokuje P1 -> takže porušuje 3. podmínku kritické sekce, nepoužitelné

Striktní střídání dvou procesů nebo vláken

– Zaveďme proměnnou *turn*, jejíž hodnota určuje, který z procesů smí vstoupit do kritické sekce. Je-li *turn* == 0, do kritické sekce může P_0, je-li == 1, pak P_1.

P_0
```
while(TRUE) {
        while(turn!=0);   /* čekej */
        critical_section();
        turn = 1;
        noncritical_section();
}
```

P_1
```
while(TRUE) {
        while(turn!=1);   /* čekej */
        critical_section();
        turn = 0;
        noncritical_section();
}
```

– **Problém:** P_0 proběhne svojí kritickou sekcí velmi rychle, *turn* = 1 a oba procesy jsou v nekritických částech. P_0 je rychlý i ve své nekritické části a chce vstoupit do kritické sekce. Protože však *turn* == 1, bude čekat, přestože kritická sekce je volná.
  - Je porušen požadavek Trvalosti postupu
  - Navíc řešení nepřípustně závisí na rychlostech procesů

- **Petersonovo řešení**
  - Funkční řešení s aktivním čekáním (while)

```
#define N2

int turn;
int interested[N];  //defaultni hodnota je 0

/* Každý proces před vstupem do KS vola enterCS
pro ověření, zda do ní může vstoupit
*/

void enterCS(int process)
{
    intotherProcess = 1 - process; //druhý proces
    interested[process] = 1;    //daný proces má zájem o KS

    // kdo jako poslední zavolá enterCS(), nastavi tak turn

    turn = process;

    //ověření, zda aktuální proces může vstoupit do KS, pokud ne, testuje

    while((turn == process)&&(interested[otherProcess] == 1)); //aktiví čekání

}
/*když proces dokonči činnost v KS, zavolá leaveCS() pro zrušení zájmu o ni a
zpřístupnění ji dalšímu procesu
*/

void leave(int process)
{
    interested[process] = 0;
}
```

- **Atomická instrukce**
  - enter_cs: task lock
    - Kopíruj lock do CPU a nastav na 1
  - baz enter_cs
    - Byl-li lock nenulový, skok na opakované testování = aktivní čekání
  - ref
    - Byl nulový -> návrat a vstup do kirtické sekce
  - leave_cs: mov lock, #0
    - vynuluj lock a odemkni kritickou sekci
  - Nutná hardwarová podpora
  - Procesor při vykonávání instrukce uzamkne datovou sběrnici
  - **Proběhne celá jako jediná operace** (nedělitelná)
  - Problém aktivního čekání je neustále testování zda může proces vstoupit do kritické sekce
  - Hrozí zde nejen plýtvaní procesorovým časem, ale také uváznuti při čekání na kritickou sekci
  - Dva procesy, různá priorita
    - H, L
    - L -> je v kritické sekci, H přijde
    - H -> je přípraven vstoupit
    - H -> nemůže vstoupit, protože L je v kritické sekci
    - H -> aktivní čekání
- **Bez aktivního čekání**
  - Jedná se o systémové volání, které uspí, nebo probudí daný proces
  - Voláním přijímá jeden parametr (id procesu)
  - V případě, že není žádná potřeba, tak se uspí

```
#define N1 = 10

int buffer[], count = 0;
void producer()
{
    while(1)
    {
        if(count==N)
        {
            sleep(producer);
        }
        buffer[count] = nextProducer;
        count++;
        if(count == 1)
        {
            wakeup(consumer);
        }
    }
}
void consumer()
{
    while(1)
    {
        if(count==0)
        {
            sleep(consumer);
        }
        bnextConsumer = buffer[count];
        count--;
        if(count == N-1)
        {
            wakeup(producer);
        }
    }
}
```

- Producent vs. Konzument – sleep/wakeup

Běžící proces	Akce	Výsledek
Konzument	Čtení count	Ak = count
Producent	Vložení položky count++ Zjisštění, že jde o první položku	count = 1 wakeup(konzument)
Konzument	? Ak == 0 ?	sleep(konzument)
Producent	Ukládá položky ? Ap == N ?	Ap = count sleep(producent)

*1 konzument, ale ještě nespí, signál je ztráceny

*2 konzument se uspí

*3 producent se uspí, dokud konzument nespotřebuje min 1. položku, konzument ale spí -> spí oba a napořád

Řešením by byla proměnná, která by kontrolovala bdělost a kontrolovala se, než by šly spát, které se nastaví, že proces nespí a nastaví wakeup

Než by proces šel spát, tak by se podíval na proměnnou, zda ten další proces spí

Semafory

- Obecný synchronizační nástroj
- Je poskytován OS a stará se o něj jádro OS
- Nachází se na začátku kritické sekce a využívají se dvě operace
    - Před vstupem do KS
    - Po vykonání KS
- Operace jsou atomické = proběhne celá
- Implementace musí zaručit, že žádné dva procesy nebudou provádět operace nad stejným semaforem současně
- Obecný semafor a binární semafor
- **Obecný semafor**
    - Jedná se o datovou strukturu obsahující celočíselný čítač a frontu čekajících procesů
    - Operace nad semaforem mohou provádět pouze funkce:
        - INIT
            - Inicializuje semafor na nezápornou hodnotu, 1 – určuje, kolik procesů muže být voláno funkcí wait
        - WAIT
            - Snižuje hodnotu čítače
            - Pokud je hodnota záporná, je proces blokován a zařazen do fronty čekajících
        - SIGNAL
            - Zvyšuje hodnotu čítače
            - Pokud je ve frontě nějaký proces, je odblokován, může vstoupit do KS

```
//Struktura semaforu
typedef struct{
    int value;  //hodnota semaforu
    struct Process * list;  //fronta procesů stojících před semaforem
} semaphore
```

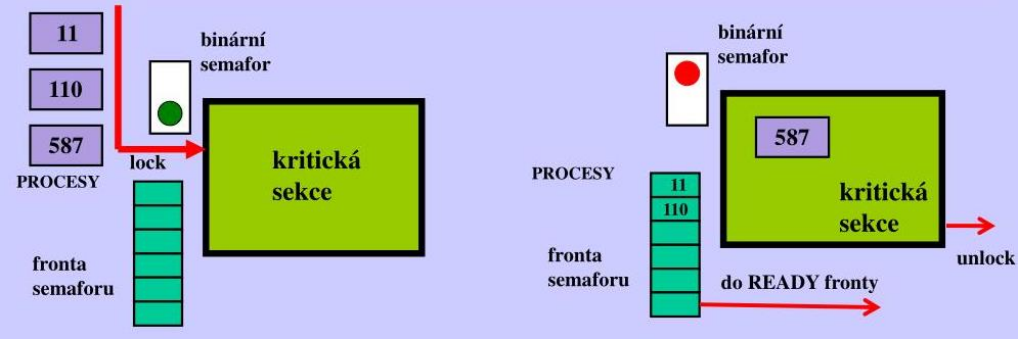
```
void wait ( semaphore & S ) {

S.value = S.value - 1;
if (S.value < 0) {  //je li treba, zablokujvolaji proces a zarad ho do
block(S.list);      //fronty pred semaforem (S.list)
 }
}

void signal ( semaphore & S ){

S.value = S.value + 1; //je-li fronta neprazdna, vyjmi proces P
if (S.value <= 0) {    // a z cela fronty a probud P
    if(S.list != NULL)
    {
        wakeup(P);
    }
 }
}
```

- **Binární semafor**
  - MUTEX
  - Semafor je implementován pomocí služeb lock a unlock
  - Lock zajistí hodnotu semaforu a nastaví jej – musí být implementováno jako nepřerušitelné
  - Pokud již je semafor nastaven (nějaký proces je v kritické sekci), uloží identifikační číslo aktivního procesu do fronty semaforu a suspenduje ho (proces suspenduje sám sebe)
  - Unlock nejprve zkontroluje stav fronty, pokud je prázdná, nastaví semafor na zelenou
  - Každý proces tak vyvolá při vstupu do kritické sekce lock, při opouštění unlock



  - Proces na začátku volá funkci lock a zjistí nastavení semaforu (TRUE/FALSE)
  - Pokud není nastaven, přepne na TRUE a vstoupí do KS
  - Pokud by byl nastaven, zařadí se do fronty čekající procesů
  - Pokud by byl nastaven, zařadí se do fronty čekajících procesů
  - Když opouští KS, volá funkci unlock a zjišťuje, jestli někdo není ve frontě a mění hodnotu semaforu, pokud ano, je vyjmut a vstoupí do KS, ale hodnota semaforu se nezmění, odblokování semaforu nastavuje poslední proces

Problémy semaforů

- **Deadlock**
  - Dva či více procesů čekající na událost, kterou může vyvolat proces, který taky čeká
- **Starvation**
  - Starnutí
  - Dva procesy si vyměňují přístup ke sdílenému prostředku, ale třetí proces se k němu nedostane
- **Aktivní zablokování**
  - Live lock
  - Procesy se navzájem snaží vyhovět, kdo půjde první
- **Inverze priorit**
  - Tři procesy, z nichž dva budou z vyšší prioritou a jeden z nižší
  - Proces z nižší prioritou pracuje se sdíleným prostředkem
  - Jeden z procesem H chce taky pracovat se sdíleným prostředkem, další proces typu H sdílený prostředek nepotřebuje, má přednost
  - L nemůže uvolnit sdílený prostředek (bude čekat na H)

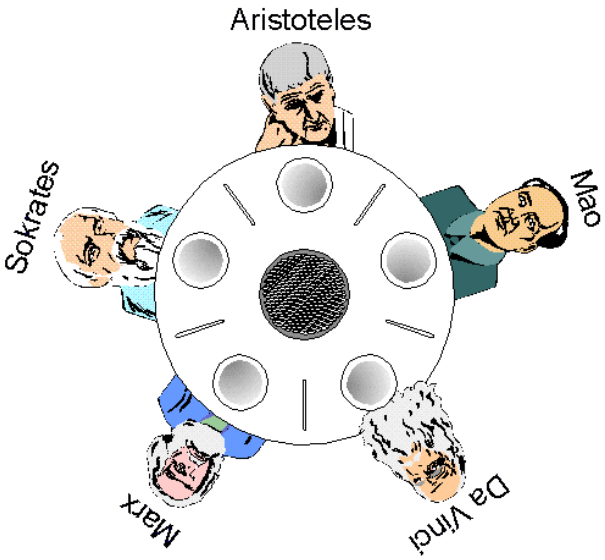
Klasické synchronizační problémy

- **Producent vs. Konzument**
  - Řeší omezenou velikost bufferu
- **Čtenáři vs. Písaři**
  - Řeší souběžné čtení a zápis dat
  - Přednost mají čtenáři
    - Žádný čtenář nečeká, není-li sdílený prostředek využíván písařem
    - Čtenář čeká na opuštění KS písařem, pokud se tak nestane, hrozí stárnutí
  - Přednost mají písaři
    - mohou modifikovat sdílený prostředek, stárnutí čtenářů

- **Večeřící filozofové**
  - Zpracovávat se můžou pouze dva procesy, které nesmí být vedle sebe
  - Buď přemýšlí, nebo večeří
  - K večeření potřebují dvě vidličky (sdílený prostředek)
  - Problém nastane, když všichni chtějí sdílený prostředek najednou
  - Řešením je reprezentovat každou vidličku jako semafor
  - Každý filozof vezme vidličku provedením funkce čekej na daný semafor
  - Odložením vidličky se provádí funkce signal
- **Spící holič**
  - Jeden holič, jedno křeslo, čekárna o velikosti n
  - Pokud nikdo není v čekárně, holič spí
  - Po obsloužení zákazníka, jde zkontrolovat čekárnu
  - Pokud je někdo v čekárně, vezme zákazníka a začne ho stříhat
  - Pokud nikdo není v čekárně, uspí se
  - Zákazník, který přijde jde rovnou k holiči
  - Pokud má volno, vzbudí holiče
  - Pokud je zaneprázdněn, jde do čekárny a zjistí stav
  - Pokud ve v čekárně volno, posadí se, pokud ne, odejde
  - Muže nastat situace, že holič spí, ale někdo je v čekárně
  - To nastane v případě, že holič jde do čekárny a zákazník k holiči a minou se, oba zjistí, že tam nikdo není, zákazník jde do čekárny a holič jde spát
  - Další problém nastane v případě, že holič stříhá a proces vejde dovnitř, zjistí, že je zaneprázdněn a mezitím ale holič dostříhá a jde se podívat do čekárny a dojde tam rychleji než zákazník, zjistí, že tam nikdo není a jde spát

Problematika večeřících filozofů podrobně

- Představme si pět filozofů, kteří tráví svůj život přemýšlením a jezením. Filozofové sdílejí jeden společný kulatý stůl, kolem kterého je pět židlí, každá pro jednoho filozofa. Uprostřed stolu je miska rýže a u kraje je rozloženo pět tyček k pojídání rýže tak, jak ukazuje následující obrázek. Když filozof přemýšlí, není v jakémkoliv spojení se svými spolustolovníky. Čas od času na filozofa přijde hlad a pokusí se získat dvě tyčky, které jsou vedle něj (ty mezi ním a jeho pravým a levým sousedem). Filozof může v jednu chvíli sebrat pouze jedno tyčku. Stejně tak nemůže tyčku vytrhnout z ruky jeho souseda, pokud ji tento právě používá. V okamžiku, kdy filozof získal obě tyčky. Může začít konzumaci ryze. Když dojí, odloží tyčky na jejich původní místa a pokračuje v přemýšlení. (Hygienické aspekty problému neuvažujeme!)
- Problém hladových filozofů je považován za klasicky synchronizační problém ne pro jeho praktické využití, ani proto, že by počítačový specialisté neměli rádi filozofy, ale proto, že je to zástupce velké třídy problémů souběžného řízení (concurrency control problems). Je to jednoduchý příklad situace, kdy různé procesy potřebují alokovat různé zdroje a může dojit k zablokování i umoření procesu.



- jednoduchým řešením je reprezentovat každou tyčku jak semafor
- každý filozof vezme tyčku provedením funkce cekej na daný semafor
- odložení tyčky se provádí funkci signal

repeat

cekej(tycka(i));

cekej(tycka(i+1 **mod** *5));*

...

jedeni

...

signal(tycka(i));

signal(tycka(i+1 **mod** *5));*

...

premysleni

...

until *false*

- řešení tohoto problému je vícero:

1. Pustit ke stolu maximálně 4 filozofy.
2. Dovolit filozofovi vzít tyčku pouze tehdy, má-li k dispozici obě.

Řešit problém asymetricky, tj. liší filozofové by nejdříve sebrali levou tyčku a potom pravou. Sudí filozofové naopak nejdříve pravou a potom levou.

- uspokojivé řešení problému musí také garantovat, že žádný z filozofu nebude umořen k smrti hlady
- je-li odstraněn deadlock (možnost zablokování) některým z výše uvedených postupu, umoření nastat nemůže