**Synchronizace procesů**

* Souběžný přístup ke sdíleným prostředkům nebo datům může způsobit jejich nekonzistenci -> nutná koordinace procesu
* Nutná komunikace mezi procesy (IPC - zasílání zpráv)

**Problém producent vs konzument**

* Producent generuje data do vyrovnávací paměti (bufferu) velikosti n
* Buffer = společná paměť
* Konzument z paměti data odebírá (čte)
* Producent aktualizuje hodnotu počítadla (count) – zvyšuje – pokud je v paměti místo -> pokud ne, tak čeká
* Konzument zároveň snižuje hodnotu počítadla (count) o jedno
* Bez správného ošetření přístupu k bufferu není zaručen správný výsledek a může dojít k uváznutí

**Kritická sekce**

Charakteristika a podmínky ošetření KS

* Část zdrojového kódu, kde dochází k přístupu ke sdíleným prostředkům (sdílení paměti nebo soubory) -> hrozí zde přístup více procesů nebo vláken
* Používá se pro zabezpečení integrity dat
* Podmínky kritické sekce:

1. Žádné 2 procesy nebo vlákna nesmí být současně uvnitř stejné kritické sekce (vzájemné vyloučení)
2. Na řešení kritické sekce nesmí mít vliv počet a rychlost CPU
3. Žádný proces mimo kritickou sekci nesmí blokovat jiný proces (trvalost postupu)
4. Žádný proces nesmí zůstat čekat nekonečně dlouho na kritickou sekci (konečné čekání)

Aktivní vs pasivní čekaní

* Aktivní čekání: proces furt běží – plýtvá procesorovým časem
* Pasivní čekání: proces se uspí začne se vykonávat jiný proces a až bude moct přistoupit tak se probudí

**Řešení kritické sekce**

* SW řešení na aplikační úrovni (je celá v režii programátora, jedná se o základní řešení s aktivním čekáním)
* HW řešení pomocí speciálních instrukcí procesoru a i zde je aktivní čekání
* SW řešení zprostředkované OS (jádro OS máme zde pasivní čekání)

1. **Vzájemné vyloučení s aktivním čekáním**
2. Zákaz přerušení

* Nejjednodušší řešení
* Zákaz všech přerušení procesy které právě vstoupily do kritické sekce
* Opětovné povolení po opuštění kritické sekce
* Pokud není povoleno přerušení nedojde k přepnutí kontextu
* Nevhodné
  + Uživatelský proces zasahuje do chodu systému
  + Pokud proces opět nepovolí přerušení může dojít k zaseknutí, zamrznutí nebo pádu systému
* Pokud je systém víceprocesorový, tak proces běžící na jiném CPU může vstoupit do kritické sekce

1. Zamykací proměnná

Void enterCS()

{

While (lock==1); //aktivní čekání

lock=1; //zamykací proměnná

}

//….

Void leaveCS()

{

lock=0;

}

* K.s budeme chtít ochránit pomocí sdílené zamykací proměnné lock s počáteční hodnotou 0 -> k.s je volná
* Když bude chtít přistoupit proces do k.s volá fci enterCS() a testuje lock
* Pokud je lock v 1 -> aktivní čekání
* Problém
  + Problém s přepínáním kontextu, kdy jeden i druhý proces načtou stejnou hodnotu = 2 procesy v k.s => nefunkční
  + Nový sdílený prostředek, nová kritická sekce, lock

1. Přesné strídání

|  |  |
| --- | --- |
| P0 | P1 |
| While(TRUE){ | While(TRUE){ |
| While(turn!=0); //čekej | While(turn!=1); //čekej |
| Critical\_section(); | Critical\_section(); |
| Turn=1; | Turn=0; |
| Noncritical\_section(); | Noncritical\_section(); |
| } | } |

* Turn říká, který proces může vstoupit do ks
* Vylepšení předchozího řešení se zamykací proměnnou
* Proces mimo ks bude blokovat druhý proces
* Aktivní čekání => plýtvání procesorového času
* Simulace:
  + Proces 0 otestoval proměnnou turn, zjistil že je 0 -> může vstoupit do ks -> vyřešil co potřeboval a nastavil trun do 1 -> pokračuje dál do n.ks -> P1 otestoval proměnnou turn, zjistil že je 1 -> může vstoupit do ks, která je u něho krátká -> dokončí práci v ks a nastaví turn na 0 -> jde do n.ks, tam se taky dlouho nezdrží (málo práce)

1. Petersonovo řešení

#define N 2

Int turn;

Int interested[N]; //defaultni hodnota je 0

//…

/\* každý proces před vstupem do k.s volá enterCS() pro ověření zda do ní může vstoupit \*/

Void enterCS(int proces)

{

Int otherProcess = 1 – process; //druhy proces

Interested[process] = 1; //dany proces ma zajem o ks

// kdo jako poslední zavolá enterCS(), nastaví tak turn!

Turn = process;

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

While((turn==process)&&(interested[otherProcess]==1));

}

//….

/\* když proces dokončí činnost v ks zavolá leaveCS() pro zrušení zájmu o ni a zpřístupní ji dalšímu procesu\*/

Void leaveCS(int process)

{

Interested[process]=0;

}

* Toto je funkční řešení s aktivním čekáním

1. Atomická instrukce

Tas-Motorola 68 000

|  |  |
| --- | --- |
| Enter\_cs: tas lock | //kopíruj lock do CPU a nastav lock=1 |
| bnz enter\_cs | /\*byl-li lock nenulový, skok na |
|  | opakované testování = aktivní čekání \*/ |
| ret | // byl-li nulový – návrat a vstup do ks |
| Leave\_cs: mov lock,#0 | //vynuluj lock a odemkni ks |
| ret |  |

* Nutná HW podpora v CPU
* Každý procesor má svou instrukci pro zpracování
* Procesor, který vykonává atomickou instrukci, uzamkne paměťovou sběrnici, čímž znemožní přistup ostatním procesům do paměti a po provedení instrukce ji pět uvolní

*Problém aktivní čekání je v neustálém testování zda může proces vstoupit do své kritické sekce*

*Hrozí zde nejen plýtvání procesorového času, ale také uváznutí při čekání na ks*

*Příklad:*

* *2 procesy s prioritou H,L*
* *Proces L běží a vstoupí do ks*
* *Objeví se proces H a chce vstoupit do ks, ale nemůže*
* *Proces H má vyšší prioritu tak L bude čekat až se H dokončí*
* *A H čeká až se dokončí L*
* *Dojde k uváznutí (inverze priority)*

1. **Sleep a Wake up**

* [Obrazek](ODKAZY/wakethefuckup.jpg)
* sleep i wakeup jsou systémová volání – jedno za úkole probudit proces a druhý je uspat -> přijímají 1 parametr (id procesu se kterým mají pracovat)
* producent zkontroluje obsah bufferu -> pokud je plný uspí se -> pokud se jedná o ukládání první informace probudí konzumenta
* konzument odebírá data z bufferu -> uspí se když není co odebírat -> kdyby odebral nejvyšší položku probudí producenta

#define N 10

Int buffer[N], count =0;

//…

Void producer(){

While(1){

If(count==N) {sleep(producer);}

Buffer[count] = nextProduced;

Count++;

If(count==1) {wakeup(consumer);}

}

}

//…

Void consumer(){

While(1){

If(count==0) {sleep(consumer);}

nextConsumed = buffer[count];

count--;

if(count==N-1) {wakeup(producer);}

}

}

* řešení problému kdy oba spí( viz. [obrazek](ODKAZY/wakethefuckup.jpg))
  + přidání kontrolního bitu pro probuzení -> toto řešení platí pouze pro 2 procesy -> pro více procesů jsou řešením semafory

1. **Semafory**

* Obecný synchronizační nástroj
* Prvně se objevil 1965
* Programový prostředek (datová struktura) – poskytována OS a spravuje jádro OS
* Nachází se na začátku ks
* Výsledkem je zda proces může vstoupit nebo nemůže do ks
* Používáme 2 atomické operace nad semafory – jedna na začátku ks druhá na konci
* Existují 2 typy semaforů: obecný a binární
* Implementace musí zaručit, že žádné 2 procesy nebudou provádět 2 operace se stejným semaforem současně

Obecný semafor

* jedná se o datovou strukturu obsahující celočíselný čítač a frontu čekajících procesů (čekajících na vstup do ks)
* operace nad semaforem mohou provádět pouze funkce
  + inid -> inicializuje semafor na nezápornou hodnotu (nejčasteji 1)
  + wait -> snižuje hodnotu čítače, pokud je hodnota záporná tak je proces blokován a nemůže vstoupit do ks, je přesunut do fronty wait
  + signal -> zvyšuje hodnotu čítače, pokud je ve frontě proces je z ní výjmut a odblokován – může vstoupit do ks
* záporná hodnota semaforu (její absolutní hodnota) udává kolik procesů stojí před semaforem (nezáporná hodnota, pak udává kolikrát je možno volat operaci)
* wait bez čekání

Struktura semaforu

Typedef struct{

Int value; //“hodnota“ semaforu

Struct process \*list; //fronta procesů stojících před semaforem

}semaphore;

* operace nad semaforem jsou pak implementovány jako nedělitelné s touto tématikou

void wait(semaphoreS){

S.value = S.value-1;

If(S.value<0) Block(S.list); /\*je-li třeba, zablokuj volající přenos a zařaď ho do fronty před semaforem (S.list) \*/

}

Void signal(semaphore S){

S.value = S.value+1;

If(S.value <=0){

If(S.list != NULL) wakeup(P); /\*je-li fronta neprázdná výjmi proces P z čela fronty a probuď P \*/

}

}

Binární semafor (Mutex)

* [obrazek](ODKAZY/mutex.jpg)
* Opět jde o datovou strukturu, ale místo čítače obsahuje boolovskou proměnnou inicializovanou na hodnotu false
* Lock(wait), unlock(signal)
* Proces na začátku zavolá funkci lock, kde zjistí jestli už není nastaven semafor, pokud není tak může vstoupit do ks a nastaví semafor
* Pokud by byl nastaven zařadí se do fronty a uspí se
* Pokud nikdo nečeká změní hodnotu semaforu (uvolní ji)

Negativa Semaforů

Dead lock(uváznutí)

* [obrazek](ODKAZY/deadlock.jpg)
* 2 či více procesů čekají na událost, kterou může vyvolat proces, který také čeká (vzájemná blokace procesu)

Starvation(stárnutí)

* 2 procesy si vyměňují přístup ke sdílenému prostředku a 3. proces se k němu nedostane (stárne)

Life block(aktivní zablokování)

* Procesy se snaží vzájemně si vyhovět (dávají si přednost, kdo bude pracovat se sdíleným prostředkem)

Priority inversion(inverze priorit)

* 3 procesy -> 2 označení H a 1 L (priority)
* Proces s prioritou L využívá sdílený prostředek -> prostředek chce využívat i proces H, ale nemůže jelikož tam je L => zablokuje se
* 3. proces typu H prostředek nepotřebuje, jenomže tím že má vyšší prioritu nedovolí L uvolnit prostředek

**Klasické synchronizační problémy**

1. Producent vs konzument

* Problém omezené vyrovnávácí paměti

1. Čtenáři a písaři

* Problém souběžného čtení a modifikace dat

1. Přednost čtenářů
   * Žádný čtenář nečeká není-li sdílený prostředek využíván písařem
   * Čtenář pouze čeká na opuštění ks písařem
   * Problém stárnutí písařů (nedostanou se k modifikaci)
2. Přednost písařů
   * Přednost písařů
   * Stárnutí čtenářů

* Transakce v databázích

1. Večeřící filozofové

Filozof představuje proces a může provádět dvě činnosti - buď obědvat nebo filozofovat. Aby mohl obědvat, musí si vzít 2 hůlky. Pokud chce filozofovat, nedrží ani jednu hůlku.

Když spolu filosofové (procesy) nekomunikují nebo komunikují nesprávně, může se každý z nich rozhodnout, že vezme například levou hůlku. Teď chce každý z nich vzít pravou hůlku, ale ta je obsazena, takže filozof nemůže ani obědvat, ani filosofovat. -> dead lock

Musí tedy buď počkat, než se hůlka uvolní, nebo hůlku položit a zkusit to později znovu.

Když se filozof nedostane po určité době k jídlu nastává starvation(stárnutí) - >Nastává například při velmi krátkých [intervalech](https://cs.wikipedia.org/w/index.php?title=%C4%8Casov%C3%BD_interval&action=edit&redlink=1) jezení a filosofování.

* Řešení
  + Přidání číšníka – číšník určí kdo si hůlky vezme

1. Spící holič

* Pokud v holičství je zákazník, zpracovává ho holič
* Když je holič zaneprázdněn zákazník si sedne do čekárny
* Když je plná čekárna jde pryč a příjde později
* Když je holičství prázdné holič spí
* Problém
  + Příjde zákazník -> jde zkontrolovat zda je holič volný -> holič zrovna dodělal práci a jde kontrolovat čekárnu (procesy se míjí) -> holič vidí, že čekárna je prázdná -> jde spát, během toho zákazník vidí, že je prázdné holičství -> jde do čekárny a spí -> oba spí dokud nepříjde další zákazník