INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA STROJNÍ

OPERAČNÍ SYSTÉMY

SYNCHRONIZACE PROCESŮ

doc. Dr. Ing. Oldřich Kodym

Ostrava 2013

ISBN 978-80-248-3053-7

Tento studijní materiál vznikl za finanční podpory Evropského sociálního fondu (ESF) a rozpočtu České republiky v rámci řešení projektu: CZ.1.07/2.2.00/15.0463, MODERNIZACE VÝUKOVÝCH MATERIÁLŮ A DIDAKTICKÝCH METOD

OBSAH

5. SYNCHRONIZACE PROCESŮ

OBSAH KAPITOLY:

Základní charakteristika a souvislosti synchronizace.

Kritická sekce.

Semafory.

Klasické problémy synchronizace.

Kritické oblasti, monitory.

MOTIVACE:

Chod operačního systému využívá mnoha standardních mechanizmů známých z jiných oblastí řízení i běžného života. Všechny události kolem nás neprobíhají „ve vzduchoprázdnu“, ale jsou více nebo méně svázány se svým okolím. Kooperující proces je proces, který může ovlivnit nebo být ovlivněn jiným procesem právě spuštěným v systému. Spolupracující procesy mohou buď přímo sdílet logický adresový prostor (jak kód, tak i data) nebo mohou sdílet data pouze prostřednictvím sdílených souborů. Konkurující přístupy k datům mohou vést k jejich narušení. Uvedeme různé mechanismy vedoucí k uspořádání přístupu spolupracujících procesů k datům tak, aby tato zůstala konzistentní.

CÍL:

Synchronizace procesů, kritická sekce

Semafory, synchronizace

1. ÚVOD

Kooperující proces je proces, který může ovlivnit nebo být ovlivněn jiným procesem právě spuštěným v systému. Spolupracující procesy mohou buď přímo sdílet logický adresový prostor (jak kód, tak i data) nebo mohou sdílet data pouze prostřednictvím sdílených souborů. Konkurující přístupy k datům mohou vést k jejich narušení. Uvedeme různé mechanismy vedoucí k uspořádání přístupu spolupracujících procesů k datům tak, aby tato zůstala konzistentní.

2. POZADÍ

Již jsme uvedli model systému, který obsahuje množství spolupracujících procesů, jež jsou spuštěny asynchronně a mohou sdílet společná data. V kapitole 3.4. jsme ilustrovali tento model pomocí omezeného bufferu (cyklického pole), který je pro OS typický.

Vraťme se k problému omezeného bufferu řešeného pomocí sdílené paměti. Je-li buffer velikosti *n*, dovoluje uložit maximálně *n*-1 položek současně. Tento nedostatek můžeme řešit několika způsoby. Jeden z nich je zavést proměnnou *citac*, jejíž hodnota se na počátku definuje jako 0 a je inkrementována při každém přidání nové položky do bufferu a dekrementována při odebrání položky.

Použitý algoritmus můžeme prezentovat zjednodušeným popisem, který odpovídá zápisu ve vyšším programovacím jazyce.

Kód producenta a příjemce by potom vypadal následovně:

Producent

```
repeat
    {vytvor dalsi polozku do nova_hodnota}
    while citac = n do nic_nedelej;
    buffer(in) := nova_hodnota;
    in := in + 1 mod n;
    citac := citac + 1;
until false;
```

Příjemce

```
repeat
    while citac = 0 do nic_nedelej;
    prijata_hodnota := buffer(out);
    out := out + 1 mod n;
    citac := citac - 1;
    {zpracuj prijatou polozku v prijata_hodnota}
until false;
```

Jak příjemce, tak producent fungují správně, jsou-li spuštěni samostatně. K chybám může dojít, poběží-li oba procesy současně. Nechť je např. hodnota proměnné *citac* 5 a producent i příjemce současně běží. V tom případě může být zcela náhodně výsledná hodnota v proměnné *citac* buď 4, 5 nebo 6! Správná hodnota je však samozřejmě jen 5 a ta by vznikla, pokud by oba procesy běžely separovaně.

Ukažme, jak k chybě může dojít. Možná implementace přirazení *citac* := *citac* + 1 může být:

```
registr1 := citac;
registr1 := registr1 + 1;
citac := registr1;
```

kde *registr1* je lokální registr CPU. Stejně tak přiřazení *citac* := *citac* - 1 může být implementováno jako:

```
registr2 := citac;
registr2 := registr2 - 1;
citac := registr2;
```

kde *registr2* je jiný lokální registr CPU.

Registr1 i *registr2* mohou být fyzicky jeden registr (akumulátor), i tehdy se však operace provedou správně díky logice CPU a mechanismu přerušení.

Současné spuštění příkazu *citac* := *citac* + 1 a *citac* := *citac* – 1 je ekvivalentní s postupným spouštěním jednotlivých příkazů nižší úrovně tak jak byly uvedeny v předem neurčeném pořadí. Jedno z možných proložení příkazů nižší úrovně může být:

```
T0: producent spouštíregistr1 := citac     {registr1 = 5}
T1: producent spouštíregistr1 := registr1 + 1  {registr1 = 6}
T2: prijemce spouští registr2 := citac     {registr2 = 5}
T3: prijemce spouští registr2 := registr2 – 1  {registr2 = 4}
T4: producent spouštícitac := registr1     {citac = 6}
T5: prijemce spouští citac := registr2     {citac = 4}
```

Výsledkem této sekvence příkazů je chybná hodnota *citac* = 4, která uvádí, že v bufferu jsou 4 položky a ve skutečnosti je jich tam po provedení obou operaci 5. V případě opačného pořadí ve vykonávání příkazů v T4 a T5, výsledkem by byla opět chyba, tentokrát *citac* = 6.

K této chybě jsme zákonitě museli dospět, protože jsme oběma procesům povolili, aby současně manipulovaly s proměnnou *citac.* Tato situace, kdy různé procesy přistupují a mění sdílená data současně a výsledek jejich činnosti závisí na pořadí, v jakém jsou jejich jednotlivé příkazy prováděny, se nazývá *souběh (race condition)*.

Pro ochranu před souběhem musíme mít jistotu, že pouze jeden proces v daném čase může měnit proměnnou *citac.* K tomu je nutná synchronizace obou procesů.

3. PROBLÉM KRITICKÉ SEKCE

Uvažujme systém obsahující *n* procesů $\{P_0, P_1, ..., P_{n-1}\}$. Každý proces má část kódu zvanou *kritická sekce,* ve které může měnit společné proměnné, updatovat tabulky, zapisovat do souboru apod. Hlavní myšlenka je, že pokud proces spouští svou kritickou sekci, nesmí žádný jiný proces mít možnost spustit ji také. Spouštění kritických sekcí procesů je *vzájemně jedinečné v čase (mutually exlusive in time)*. Problémem kritické sekce je vytvořit protokol, který procesům umožní spolupracovat. Každý proces musí mít právo provést svou kritickou sekci. Část kódu implementující tento dotaz se nazývá *vstupní (entry)* sekce. Kritická sekce pak může být ukončena *ukončovací (exit)* sekcí. Zbývající kód se někdy nazývá *zbývající (remainder) sekce.*

```
repeat
    vstupni sekce
    kriticka sekce
    ukoncovaci sekce
    zbyvajici sekce
until false
```

Řešení problému kritické sekce musí zohledňovat následující 3 požadavky:

- **vzájemnou jedinečnost (mutual exclusion):** Jestliže proces P_i vykonává svou kritickou sekci, žádný jiný proces ji vykonávat nesmí.

- **progress:** Jestliže u žádného procesu neprobíhá jeho kritická sekce a existují nějaké procesy, které chtějí svou kritickou sekci spustit, pak pouze ty procesy, které vykonávají zbývající sekci, se mohou účastnit rozhodnutí, která kritická sekce kterého procesu bude spuštěna a tento výběr nesmí být neomezeně odložen.
- **omezené čekání (bounded waiting):** Musí existovat omezení v počtu časových kvant mezi odesláním požadavku na výkon kritické sekce procesem a umožněním tohoto výkonu.

V následujících kapitolách ukážeme řešení problému kritické sekce, která zohledňují výše uvedené vlastnosti.

3.1 Řešení pro dva procesy

V této kapitole zúžíme svoji pozornost na algoritmy aplikovatelné pouze pro dva procesy v čase. Procesy budou označeny P0 a P1. V obecném označení procesů užijeme Pi a Pj, tzn. j = 1 – i.

3.1.1 Algoritmus 1

První přístup spočívá ve sdílené celočíselné proměnné *otacka*, která bude moci nabývat hodnot pouze 0 a 1. Jestliže *otacka* = *i*, potom proces Pi může spouštět svou kritickou sekci, jak ukazuje následující výpis.

```
repeat
    while otacka <> i do nic_nedelej;
    kriticka sekce
    otacka := j;
    zbyvajici sekce
until false;
```

Výpis 1 - Struktura procesu Pi v algoritmu 1

Toto řešení zajišťuje, že pouze jeden proces ve stejné době může mít spuštěnu kritickou sekci. Je-li *otacka* = 0, proces P1 svou kritickou sekci spustit nemůže (viz výše).

3.1.2 Algoritmus 2

Problém algoritmu 1 je v tom, že nedrží dostatečné informace o stavu každého procesu, rozhoduje pouze, který proces může spustit kritickou sekci. Řešení toho problému tkví v nahrazení proměnné *otacka* následujícím polem:

```
var priznak: array (0..1) of boolean;
```

Prvky pole *příznak* jsou na počátku inicializovány do hodnoty *false*. Je-li *P(i)* = *true*, potom proces Pi je připraven vstoupit do své kritické sekce. Strukturu tohoto algoritmu pro proces Pi ukazuje následující výpis:

```
repeat
    priznak(i) := true;
    while priznak(j) do nic_nedelej;
    kriticka sekce
    priznak(i) := false;
    zbyvajici sekce
until false;
```

Výpis 2 - Struktura procesu Pi v algoritmu 2

Je-li proces Pi připraven vstoupit do své kritické sekce, nastaví hodnotu *priznak(i)* na *true* a kontroluje *priznak(j)*, zdali proces Pj neprovádí právě svou kritickou sekci. Pokud ano, Pi musí čekat, až bude kritická sekce procesu Pj dokončena a potom může spustit svou. Po jejím

ukončení nastaví hodnotu *priznak(i)* na *false*, čímž dává na vědomí, že další proces může začít se svou kritickou sekcí (je-li takový).

U tohoto algoritmu je požadavek vzájemné jednoznačnosti vyřešen, bohužel na rozdíl od požadavku progresivnosti. Pro ilustraci problému užijme nesledující spouštěnou sekvencí:

```
T0: P0 nastavuje priznak(0) = true
T1: P1 nastavuje priznak(1) = true
```

V tomto případě nastává u obou procesů zacyklení v cyklu **while**.

3.1.3 Algoritmus 3

Kombinuje myšlenku algoritmu 1 i 2 a výsledkem je řešení problémů kritické sekce, které splňuje všechny tři dříve uvedené požadavky. Oba procesy sdílejí 2 proměnné:

```
var priznak: array (0..1) of boolean;
    otacka: 0..1;
```

Na počátku jsou hodnoty proměnných inicializovány *priznak(0) = priznak(1) = false*, hodnota proměnné *otacka* je nepodstatná. Strukturu algoritmu 3 pro proces Pi ukazuje následující výpis:

```
repeat
    priznak(i) := true;
    otacka := j;
    while (priznak(j) and otacka = j) do
        nic_nedelej;
    kriticka sekce
    priznak(i) := false;
    zbyvajici sekce
until false;
```

Výpis 3 - Struktura procesu Pi v algoritmu 3

Před vstupem do kritické sekce procesu Pi se nejprve nastaví proměnná *priznak(i)* na hodnotu *true* a potom tvrdí, že druhý proces nechce vstoupit do kritické sekce (*otacka* := *j*). Jestliže oba procesy se pokouší současně spustit své kritické sekce, proměnná *otacka* určí, který z obou procesů může spustit svou kritickou sekci jako první. Rozhodne o tom poslední přiřazení hodnoty proměnné *otacka*, předcházející bude přepsáno.

K implementaci bodu 1 (vzájemná jednoznačnost) je možno podotknout, že mají-li být spuštěny současně kritické sekce obou procesu P0 i P1, je k tomu zapotřebí, aby *priznak(0) = priznak(1) = true* ovšem proměnná *otacka* může nabýt pouze hodnoty 1 nebo 0, ne však obou najednou, takže k současnému spuštění kritických sekcí obou procesů dojít nemůže. Pro jeden z nich je splněna podmínka cyklu **while**, budiž to proces Pi, a dostává se do čekací smyčky až do doby, kdy se *priznak(j)* nastaví do hodnoty *false*, k čemuž dojde až po ukončení kritické sekce procesu Pj.

K implementaci bodu 2 a 3 poznamenejme, že procesu Pi může být zabráněno vstoupit do kritické sekce pouze, pokud je zadržen cyklem **while** s podmínkou *priznak(j)* je *true* a *otacka = j* a tento cyklus je jen jeden. Pokud se Pj nechystá vstoupit do kritické sekce, je hodnota *priznak(j) = false* a Pi může vstoupit do kritické sekce. Pokud proces Pj nastavil *priznak(j) = true*, potom je *otacka = j* nebo *otacka = i*. Jestliže *otacka = i* potom proces Pi vstoupí do kritické sekce. Jestliže *otacka = j* potom svou kritickou sekci vykoná proces Pj. Presto po vykonání kritické sekce procesem Pj bude hodnota *priznak(j)* nastavena na *false*, čímž je umožněno procesu Pi opustit cyklus **while** a vykonat svou kritickou sekci. Jestliže proces Pj nastaví hodnotu *priznak(j)* na *true* musí nastavit proměnnou *otacka* na *i*. Protože Pi nemůže změnit hodnotu proměnné *otacka* zatímco je blokován cyklem **while**, Pi vstoupí do kritické sekce (progress) maximálně po jednom průběhu kritické sekce procesu Pj (omezené čekání).

3.2 Řešení pro více procesů

Algoritmus 3 řeší korektně problém kritické sekce pro 2 procesy. Nyní rozšíříme řešení na *n* procesů. Tento algoritmus je znám pod názvem *bakery algorithm (pekařův algoritmus)* a je postaven na plánovacím algoritmu, který se užívá v pekařství, skladech zmrzliny, masnách a všude jinde, kde je třeba z chaosu udělat pořádek. Tento algoritmus byl vyvinut pro distribuované prostředí, ale nyní se budeme věnovat pouze těm jeho aspektům, které se týkají prostředí koncentrovaného.

Při vstupu do obchodu dostane každý zákazník číslo. Zákazník s nejmenším číslem bude obsluhován nejdříve. Pekařův algoritmus nemůže garantovat, že dva procesy neobdrží stejné číslo. V tomto případě je obsloužen dříve algoritmus s menším "jménem", tzn., jestliže procesy *Pi* a *Pj* dostaly stejné číslo a $i < j$, potom Pi bude obsloužen nejdříve. Protože jména procesů jsou jedinečná a absolutně uspořádaná, algoritmus je kompletně deterministicky.

Datové struktury nutné pro implementaci pekařova algoritmu jsou:

```
var vyber: array (0..n-1) of boolean;
    cislo: array (0..n-1) of integer;
```

Na počátku jsou obě pole inicializována na hodnoty *false* a 0. Pro zjednodušení zápisu algoritmu definujme následující vztahy:

```
(a,b) <(c,d) <=> ((a <c) or (a = c)) and (b <d).
max(a0, a1, ..., an-1) = k <=> k >= ai pro i = 0, 1, ... , n-1
```

Strukturu procesu Pi ukazuje výpis 4.

Pro ověření správnosti pekařova algoritmu musíme nejprve ukázat, že jestliže Pi provádí svou kritickou sekci a Pk (k <> i) má přiděleno číslo *cislo(k)*, potom (*cislo(i)*,*i*) < (*cislo(k)*,*k*).

Výše uvedené tvrzení dokazuje splnění požadavku vzájemné jednoznačnosti u pekařova algoritmu. Dále uvažujme, že Pi vykonává svou kritickou sekci a Pk tou dobou chce spustit svou kritickou sekci. Když Pk vstoupí do druhého cyklu **while** s tím, že j = i, podmínky jsou:

```
cislo(i) <> 0
(cislo(i),i) < (cislo(k),k).
```

Tyto podmínky vedou cyklení procesu Pk v cyklu **while** dokud proces Pi bude provádět svou kritickou sekci.

Pro doložení vlastnosti progresivnosti a omezeného čekání je vhodné si uvědomit, že každý proces spouští svou kritickou sekci systémem first-come first-serverd.

```
repeat
    vyber(i) := true;
    cislo(i) := max(cislo(0), ... cislo(n-1))+ 1;
    vyber(i) := false;
    for j = 0 to n - 1 do begin
        while vyber(j) do nic_nedelej;
        while cislo(j) <> 0 and (cislo(j),j) < (cislo(i),i) do
            nic_nedelej;
    end;
    kriticka sekce
    cislo(i) := 0;
    zbyvajici sekce
until false;
```

Výpis 4 - Pekařův algoritmus pro proces Pi

4. SYNCHRONIZAČNÍ HARDWARE

Stejně jako jinde, i v synchronizaci procesů může hardware zjednodušit softwarovou implementaci a zvýšit efektivitu systému. V této kapitole ukážeme některé hardwarové instrukce, které jsou implementovány v mnoha systémech, a ukážeme, v čem přispívají k řešení problému kritické sekce. Nevýhodou tohoto řešení je obvykle skutečnost, že determinuje použitelný algoritmus.

Jednoduchým řešením problému kritické sekce by bylo zakázat ošetření přerušení v době, kdy jsou modifikovány systémové proměnné. Potom bychom si mohli být jisti, že libovolná sekvence programu bude celá vykonána bez nuceného přerušení. Žádná jiná instrukce jiného procesu by nebyla spuštěna, takže by také nemohlo dojít k nečekané modifikaci systémových proměnných.

Toto řešení však není vždy proveditelné. Například u multiprocesorového systému by to mohlo vést ke značným časovým ztrátám, dejme tomu v případě zprávy, která prochází všemi procesory. Taková zpráva by musela na každém procesoru čekat na případné dokončení kritické sekce, jež na něm může běžet a efektivita systému by klesla.

Mnoho počítačů provozuje hardwarovou instrukci, která umožňuje testovat a měnit obsah slova a další, která vymění obsah dvou slov. Těchto speciálních instrukcí můžeme využít k relativně jednoduchému řešení problému kritické sekce. Spíše než diskusemi o implementaci těchto instrukcí na jednotlivých počítačích se věnujme pochopení jejich fungování. Instrukce *Test-and-Set* může být implementována např. takto:

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

Základní charakteristikou funkce *Test-and-Set* je, že je spouštěna nepřerušitelně. Tedy, jestliže dvě funkce *Test-and-Set* jsou spuštěny souběžně (na dvou různých procesorech), budou spuštěny postupně v libovolném pořadí.

Pokud počítač podporuje funkci *Test-and-Set*, může implementovat časovou jedinečnost za pomoci logické proměnné *zamek*, inicializované do počáteční hodnoty *false*. Výpis 5 ukazuje strukturu procesu Pi.

Instrukce *Vymena*, která automaticky zamění obsah dvou slov. Stejně jako instrukce *Test-and-Set* je i instrukce *Vymena* spouštěna v *nepřerušitelném atomickém* režimu. Definice instrukce *Vymena* může být následující:

```
procedure Vymena(var a, b: boolean);
var docasna: boolean;
begin
    docasna := a;
    a := b;
    b := docasna;
end;
```

Řešení problému kritické sekce pomoci instrukce *Test-and-Set*:

```
repeat
    while Test-and-Set(zamek) do nic_nedelej;
    kriticka sekce
    zamek := false;
    zbyvajici sekce
until false;
```

Výpis 5 - Implementace časové jedinečnosti pomocí Test-and-Set

Podporuje-li počítač instrukci *Vymena* může být časová jedinečnost implementována tak, jak ukazuje výpis 6. Globální logická proměnná *zamek* je na počátku inicializována na *false* a každý proces má svou vlastni lokální proměnnou *klic*.

```
repeat
     klic:= true;
     repeat
          Vymena(zamek, klic);
     until klic = false;
     kriticka sekce
     zamek := false;
     zbyvajici sekce
until false;
```

Výpis 6 - Implementace časové jedinečnosti pomocí instrukce Vymena

Oba uvedené algoritmy nezohledňují požadavek omezeného čekání. Následující algoritmus, který bude pracovat na základě instrukce Test-and-Set bude splňovat všechna kritéria korektního řešení problému kritické sekce.

Algoritmus bude využívat následujících datových struktur:

```
var cekani: array (0 .. n-1) of boolean;
zamek: boolean
```

Všechny proměnné jsou inicializovány na počátku do hodnoty *false*.

```
var j: 0..n-1;
    klic: boolean;
repeat
     cekani(i):= true;
     klic := true;
     while cekani(i) and klic do klic := Test-and-Set(zamek);
     cekani(i):= false;
     kriticka sekce
     j := i + 1 mod n;
     while (j <> i) and (not cekani(j)) do j := j + 1 mod n;
     if i = j then zamek := false
             else cekani(j) := false;
     zbyvajici sekce
until false;
```

Výpis 7 - Korektní řešení problému kritické sekce pomocí instrukce Test-and-Set

Řešení problému časové jednoznačnosti: Proces Pi může začít vykonávat svou kritickou sekci pouze jestliže *cekani(i) = false* nebo když *klic = false*. *Klic* může získat hodnotu *false* pouze jestliže dojde ke spuštění instrukce *Test-and-Set*. První proces, který spustí instrukci *Test-and-Set* získá hodnotu *klic = false*, všechny ostatní musí čekat. Proměnná *cekani(i)* může nabýt hodnoty *false* pouze jestliže jiný proces opustí svou kritickou sekci. Pouze pro jednu hodnotu *i* je *cekani(i) = false*, čímž je časová jednoznačnost zajištěna.

Řešení problému progresivnosti: Proces, který ukončí svou kritickou sekci buď nastaví *klic* na *false* nebo *cekani(j)* na *false*. Obě dvě přiřážení vedou k možnosti vstupu dalšího čekajícího procesu do jeho kritické sekce z důvodů uvedených v předcházejícím odstavci.

Řešení problému omezeného čekání: Jestliže proces ukončí svou kritickou sekci, prohlíží se cyklicky pole *cekani* v pořadí ($i + 1$, $i + 2$, ..., $n - 1$, 0, 1, ..., $i - 1$). Tím je nalezen první proces v uvedeném pořadí, který je ve vstupní sekci (jeho *cekani(j) = true*), jako proces, jehož kritická sekce bude spuštěna. Na libovolný z čekajících procesů se tedy dostane po maximálně $n - 1$ pokusech.

5. SEMAFORY

Řešeni problému kritické sekce tak, jak bylo uvedeno v předcházející kapitole je sice velmi jednoduché pro pochopení, nicméně rozšiřovat tyto algoritmy pro komplexnější problémy není jednoduché. Tam je již třeba užít jiného synchronizačního nástroje – *semaforu*. Semafor *S* je celočíselná proměnná, jejíž hodnota může být změněna, kromě počáteční inicializace, pouze dvěma standardními *atomickými* operacemi: *cekej* a *signal*. Klasická definice obou operací je:

```
cekej(S):
    while S <= 0 do nic_nedelej;
    S := S - 1;

signal(S):
    S := S + 1;
```

Modifikace celočíselných proměnných pomoci operaci *cekej* a *signal* je jedinečná a nepřerušitelná, tj. pokud některý proces modifikuje hodnotu semaforu, nesmí žádný jiný proces ve stejnou dobu modifikovat hodnotu stejného semaforu. Běh operací nesmí být přerušen jiným procesem.

Implementaci operací ukážeme později, nyní se věnujme problému praktického užití semaforu.

5.1 Užití

Je možno užit semafory pro řešení problému kritické sekce *n*-procesu. *N* procesů sdílí semafor *vzajed* (zkratka ze *vza*jemna *jed*inecnost), inicializovaný do hodnoty 1. Každý proces *Pi* je organizován jak je vidět na Obr. 60.

```
repeat
    cekej(vzajed);
    kriticka sekce
    signal(vzajed);
    zbyvajici sekce
until false;
```

Výpis 8 - Řešení problému kritické sekce pomocí semaforu

Semafory lze užít pro řešení různých synchronizačních problémů. Uvažujme např. dva současně probíhající procesy: P1 s instrukcí I1 a P2 s instrukcí I2. Nechť instrukce I2 může být spuštěna pouze po dokončení instrukce I1. Řešení můžeme implementovat s pomocí sdílené proměnné *synch* inicializované na počátku do 0 a vložit do programu procesů P1 a P2 tyto instrukce:

```
P1:
    I1;
    signal(synch);
P2:
    cekej(synch);
    I2;
```

Protože proměnná *synch* je na začátku inicializována do hodnoty 0, P2 může spustit instrukci S2 pouze, pokud P1 vykoná instrukci *signal*(*synch*), která se spustí po dokončení instrukce S1.

5.2 Zablokování a umoření

Implementace semaforu s frontou čekajících může vést k situaci, kdy dva nebo více procesů neomezeně čekají na událost, kterou může vyvolat pouze některý z čekajících procesů. Jádro

problému je ve spouštění operace *signal*. O procesech, které dosáhly výše uvedeného stavu, hovoříme jako o *zablokovaných.*

Pro ilustraci *zablokování* procesu uvažujme systém obsahující dva procesy. P0 a P1, a každý z nich přistupuje ke dvěma semaforům *S* a *Q* a nastaví jejich hodnotu na 1:

```
   P0              P1
cekej(S);       cekej(Q);
cekej(Q);       cekej(S);
    .               .
    .               .
    .               .
signal(S);      signal(Q);
signal(Q);      signal(S);
```

Uvažujme, že *P0* spustí *cekej(S)* a bezprostředně potom *P1* spustí *cekej(Q)*. Potom když *P0* spustí *cekej(Q)*, musí čekat dokud *P1* nespustí *signal(Q)*. Současně s tím, když *P1* spustí *cekej(S)*, musí čekat, dokud *P0* nespustí *signal(S)*. Ani jedna z těchto operací však nemůže být provedena a procesy *P0* a *P1* jsou zablokovány.

Říkáme, že množina procesů je ve stavu zablokována, jestliže každý proces v množině čeká na událost, kterou může vyvolat pouze jiný proces z téže množiny. Událost, se kterou se zde budeme stýkat zejména, je obsazení zdroje a jeho uvolnění. K zablokování procesu mohou vést i jiné situace jak ukážeme v kapitole 7. V té kapitole také uvedeme možné mechanismy vedoucí k řešení problému zablokování.

Další problém svázaný se zablokováním procesu je *umoření (starvation)* procesu, situace kdy proces čeká neomezeně dlouho na semaforu. Neomezené zablokování nebo umoření procesu může nastat, jestliže přidáváme nebo odebíráme proces ze seznamu asociovaného s daným semaforem v LIFO pořadí.

5.3 Binární semafory

Semafory definované v předešlé kapitole jsou známy jako *čítací* semafory, jejichž hodnota není nijak omezena. *Binární* semafor je semafor s celočíselnou hodnotou, která může nabývat pouze hodnot 0 a 1. Binární semafor může být implementován jednodušeji než semafor čítací, který vyžaduje hardwarovou podporu.

Ukážeme nyní, jak může být čítací semafor implementován pomocí binárních. Nechť *S* je čítací semafor. Pro jeho implementaci pomocí binárních semaforů potřebujeme následující datové struktury:

```
var S1, S2, S3 : binarni_semafor;
    C : integer;
```

Počáteční inicializace proměnných je *S1* = *S3* = 1, *S2* = 0. Proměnná *C* je nastavena do počáteční hodnoty čítacího semaforu *S*.

Operace *cekej* na čítacím semaforu *S* může být implementována následovně:

```
cekej(S3);
cekej(S1);
C := C – 1;
if C < 0 then begin
    signal(S1);
    signal(S2);
end else signal(S1);
signal(S3);
```

Operace *signal* na čítacím semaforu *S* může být implementována následovně:

```
cekej(S1);
```

```
C := C + 1;
if C <= 0 then signal(S2);
signal(S1);
```

Semafor *S3* se nijak nepodílí na operaci *signal(S)*, ten svou hodnotu mění pouze při operaci *cekej(S)*.

6. KLASICKÉ PROBLÉMY SYNCHRONIZACE

V této kapitole ukážeme některé synchronizační problémy, které jsou významné zejména proto, že jsou to ukázky z velké třídy problémů vyplývajících z problémů konkurenčního přístupu. Tyto problémy se užívají k testování téměř každého nově navrhovaného synchronizačního schématu. K synchronizaci v našem řešení užijeme semafory.

6.1 Problém omezeného bufferu

Problém omezeného bufferu byl poprvé uveden v kapitole 6.1. Je často uváděn k ilustraci síly synchronizačních primitiv. Ukážeme zde hlavní strukturu tohoto schématu bez komentování jakékoli konkrétní implementace.

Předpokládejme, že společná oblast (pool) obsahuje *n* bufferů, každý je schopen uchovávat jednu položku. Semafor *vzajed* zajišťuje vzájemně jedinečný přístup do společné oblasti bufferů a je na počátku inicializován do hodnoty 1. Semafory *prazdny* a *plny* počítají počet prázdných a plných bufferů. *Plny* je na počátku inicializován na 0, *prazdny* na *n*. Kód producenta je vidět na Obr. 60, kód příjemce na Obr. 61. Povšimněme si symetrie mezi producentem a příjemcem. Tento kód můžeme interpretovat tak, že producent produkuje plné buffery pro příjemce, nebo že příjemce vytváří prázdné buffery pro producenta.

6.2 Problém zapisovače a snímače

Datové objekty (jako např. soubor nebo záznam) mohou být sdíleny různými konkurujícími procesy. Některé z těchto procesů mohou chtít pouze číst obsah sdíleného objektu, zatímco jiné mohou chtít updatovat sdílený objekt (tedy číst i do něj zapisovat). Budeme rozlišovat mezi těmito dvěma typy procesů a ty, které budou chtít pouze číst ze sdíleného objektu, označíme jako *snímače (readers)* a zbylé jako *zapisovače (writers)*. Pochopitelně budou-li současně přistupovat ke sdílenému objektu dva snímače, žádný problém nenastane. Pokud by ovšem se sdílenými daty pracoval zapisovač a zároveň jiný proces (ať už snímač nebo také zapisovač), k potížím by dojít mohlo.

Pro zamezení zmiňovaným problémům je třeba zaručit zapisovačům exkluzivní přístup ke sdíleným datovým objektům. Tento typ synchronizace je uváděn jako problém *snímač–zapisovač (reader–writer)*. Problém snímač–zapisovač má mnoho podob zahrnující různé priority.

Nejjednodušší verze, známá jako *první problém snímač–zapisovač* požaduje, aby žádný snímač se nedostal do stavu *čekající*, ledaže by právě se sdíleným objektem pracoval zapisovač. Jinak řečeno, žádný snímač by neměl čekat na jiný snímač, protože čekat případně *musí* zapisovač.

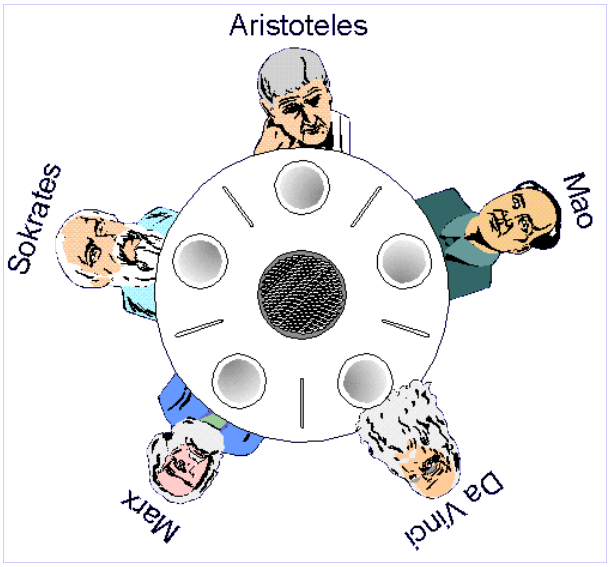
Druhý problém snímač–zapisovač vyžaduje, aby jakmile je zapisovač připraven psát do sdíleného objektu musí to vykonat nejrychleji jak je možné. Jinak řečeno, pokud zapisovač čeká na přístup k objektu, žádný jiný snímač nesmí začít s objektem pracovat.

Poznamenejme, že řešení výše uvedených problémů může vést k umoření procesu. V prvním případě může dojít k umoření zapisovače, v druhém k umoření snímače. Z tohoto důvodu byly sestaveny další algoritmy, které řeší i problém umoření.

6.3 Problém hladových filozofů (The Dining-Philosophers Problem)

Představme si pět filozofů, kteří tráví svůj život přemýšlením a jedením. 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 hůlek 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 hůlky, které jsou vedle něj (ty mezi ním a jeho pravým a levým sousedem). Filozof může v jednu chvíli sebrat pouze jednu hůlku. Stejně tak nemůže hůlku vytrhnou z ruky jeho souseda, pokud ji tento právě používá. V okamžiku, kdy filozof získal obě hůlky, může zahájit konzumaci rýže. Když dojí, odloží hůlky na jejich původní místa a pokračuje v přemýšlení. (Hygienické aspekty problémů v tomto konkrétním případě neuvazujeme!)

Problém hladových filozofů je považován za klasický synchronizační problém ne pro jeho praktické využití, ani proto, že by počítačoví 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 procesů.



Obrázek 1 - Problém hladových filozofů

Jednoduchým řešením je reprezentovat každou hůlku jak *semafor*. Každý filozof vezme hůlku provedením funkce *cekej* na daný semafor. Odložení hůlky se provádí funkcí *signal*. Sdílená data mohou být tedy reprezentována datovou strukturou:

```
var hulka: array (0..4) of semafor;
```

kde všechny prvky pole jsou na počátku iniciovány do hodnoty 1.

Strukturu *i*-tého filozofa ukazuje výpis 9. Toto řešení garantuje, že žádní dva sousední filozofové nebudou konzumovat současně, ovšem za cenu možného zablokování filozofa. Uvažme, že všichni filozofové by dostali hlad současně. Tehdy by nejprve všichni uchopili hůlku, kterou mají po levé ruce, čímž by všechny semafory přešly do stavu 0. Každý filozof, který by pak chtěl uchopit hůlku po své pravici, se dostává do nekonečné čekací smyčky.

```
repeat
    cekej(hulka(i));
    cekej(hulka(i+1 mod 5));
        ...
        jedeni
        ...
```

```
        signal(hulka(i));
        signal(hulka(i+1 mod 5));
             ...
             premysleni
             ...
until false
```

Výpis 9 - Struktura i-tého filozofa

Řešení tohoto problému je vícero:

- Pustit ke stolu maximálně 4 filozofy.
- Dovolit filozofovi vzít hůlku pouze tehdy, má-li k dispozici obě.
- Řešit problém asymetricky, tj. liší filozofové by nejdříve sebrali levou hůlku a potom pravou. Sudí filozofové naopak nejdříve pravou a potom levou.
- Řešení za užití monitoru si ukážeme v patřičné kapitole.

Uspokojivé řešení problému musí také garantovat, že žádný z filozofů nebude umořen k smrti hlady. Je-li odstraněn deadlock (možnost zablokování) některým z výše uvedených postupů, umoření nastat nemůže.

7. KRITICKÉ OBLASTI

Ačkoliv semafory jsou vhodným a efektivním řešením problémů synchronizace procesů, jejich nesprávné použití může vyvolat nejrůznější chyby v časování, které se velmi těžko odhalují, protože nastávají pouze „někdy“ – tehdy jsou-li v určitém okamžiku spuštěny určité sekvence určitých procesů.

V kapitole 2 jsme ukázali jeden příklad chyb, které mohou nastat při řešení problému producenta a příjemce. V tomto případě dochází k problému v časování poměrně zřídka a řešení pomoci semaforu je jednoduché a účinné.

Bohužel, chyby v časování mohou nastat i při užití semaforu. K ilustraci, jak k tomu může dojít, je třeba si připomenout, jak je pomocí semaforu řešen problém kritické sekce. Všechny procesy sdílí proměnnou semafor *vzajed*, která je na počátku iniciována do hodnoty 1. Každý proces musí spustit funkci *cekej(vzajed)* před tím, než vstoupí do kritické sekce a funkci *signal(vzajed)* poté, co z kritické sekce vystoupí. Jestliže není tato sekvence zachována, mohou být dva procesy současně ve svých kritických sekcích.

Uvažme potíže, které mohou při tomto řešení vyvstat. Poznamenejme, že k těmto potížím dochází pouze v případě, když se některý proces chová nekorektně. K tomu může dojít buď chybou programu, nebo nespolupracuje-li proces s ostatními.

1. Uvažujme, že by některý proces volal funkce *cekej* a *signal* v opačném pořadí, tj.:

```
signal(vzajed);
       ...
kriticka sekce
       ...
cekej(vzajed);
```

Díky této chybě může nastat situace, že více procesů sdílejících semafor *vzajed* bude současně ve své kritické sekci. Odhalení této chyby je možné pouze v případě, ze taková situace nastane (tj. že více procesů bude současně aktivních ve svých kritických sekcích).

2. Nechť jeden proces obsahuje tuto sekvenci příkazů:

```
cekej(vzajed);
```

```
        ...
kriticka sekce
        ...
cekej(vzajed);
```

V tomto případě může nastat zablokování (deadlock) procesu.

3. Uvažme, že některý proces vynechá buď funkci *cekej* nebo *signal* nebo obě dvě. V tom případě nastává buď porušení vzájemné jednoznačnosti nebo deadlock.

Tyto výše uvedené příklady ukazují různé chyby, které mohou velmi jednoduše nastat, pokud je nesprávně pracováno se semafory při řešení problemu kritické sekce. Jiné problémy jsme již diskutovali v předcházející kapitole.

Ukážeme si první řešení takovýchto problémů, které spočívá ve vytvoření synchronizačního mechanismu vyšší úrovně - *kritických oblastí (critical regions)*, někdy též nazývaných *podmíněně kritické oblasti (conditional critical regions)*. V následující kapitole ukážeme další základní řešení, řešení pomoci *monitorů*.

V prezentaci obou synchronizačních konstrukcí budeme uvažovat, že proces obsahuje lokální data a sekvenční program, který s nimi pracuje. Lokální data mohou být modifikována pouze sekvenčním programem téhož procesu. To znamená, že žádný proces nemůže přímo modifikovat lokální data jiného procesu. Procesy mohou nicméně sdílet data globální.

Synchronizační konstrukce vyšší úrovně – *kritická oblast* vyžaduje nějakou proměnnou *p* typu *T*, která bude sdílena různými procesy. Deklarace:

```
var p: shared T;
```

K proměnné *p* je možno přistoupit pouze uvnitř *oblasti* následujícím příkazem

```
region p when B do S;
```

Tato konstrukce znamená, že dokud je spuštěn příkaz *S*, žádný jiný proces nemůže přistoupit k proměnné *p*. Proměnná *B* je logická proměnná, která řídí vstup do kritické oblasti. Když proces hodlá vstoupit do oblasti, je zkontrolována hodnota proměnné *B*. Jestliže je *true*, příkaz *S* může být spuštěn. Je-li *false*, proces se musí vzdát vstupu do oblasti a vyčkat, dokud mu nebude proměnná *B* příznivě nakloněna a žádný jiny proces nebude v kritické oblasti asociované s proměnnou *p*.

To jest, jestliže dva příkazy:

```
region p when true do S1;
region p when true do S2;
```

jsou současně spuštěny v různých sekvenčních procesech, je výsledek ekvivalentní posloupnosti příkazů „*S1* následuje za *S2*“ nebo „*S2* následuje za *S1*“.

Konstrukce *kritické oblasti* zabraňuje vzniku některých jednoduchých programátorských chyb spojených s řešením problému kritické sekce pomoci *semaforu*. Nepředchází všem chybám, ale zmenšuje jejich počet. Jestliže totiž chyba nastane v logice programu, výsledná sekvence událostí nemusí být právě nejjednodušeji postižitelná.

Konstrukce *kritické oblasti* může být efektivně využita k řešení některých synchronizačních problémů. Pro ilustraci takového řešení využijeme problém omezeného bufferu. Prostor pro buffer i ukazatele jsou definovány jako:

```
var buffer: shared record
        pool: array (0..n-1) of polozka;
        citac, in, out : integer;
        end;
```

Producent uloží do bufferu hodnotu v položce *nova_hodnota* následující sekvencí příkazů:

```
region buffer when citac < n
do begin
    pool(in) := nova_hodnota;
    in := in + 1 mod n;
    citac := citac + 1;
end;
```

Příjemce odebere z bufferu položku a uloží ji do proměnné *prijata_hodnota* následující sekvencí příkazů:

```
region buffer when citac > 0
do begin
    prijata_hodnota := pool(out);
    out := out + 1 mod n;
    citac := citac - 1;
end;
```

Ukažme si nyní možnou implementaci kritické oblasti. S každou sdílenou proměnnou jsou asociovány následující proměnné:

```
var vzajed, prvni_cekani, druhe_cekani : semafor;
    prvni_citac, druhy_citac : integer;
```

Semafor *vzajed* je na počátku inicializován do 1, *prvni_cekani* a *druhe_cekani* do 0, stejně jako celočíselné proměnné *prvni_citac* a *druhy_citac*.

Vzájemně jedinečný přístup do kritické sekce je řízen semaforem *vzajed*. Jestliže proces nemůže vstoupit do své kritické sekce, neboť logická proměnná *B* je *false*, proces čeká na semaforu *prvni_cekani*. Nežli je nastavena hodnota proměnné *B* na *true*, je proces přesunut na semafor *druhe_cekani*. V proměnných *prvni_citac* a *druhy_citac* je sledován počet procesů čekajících na semaforech *prvni_cekani* a *druhe_cekani*.

```
cekej(vzajed);
while not B do begin
    prvni_citac := prvni_citac + 1;
    if druhy_citac > 0
        then signal(druhe_cekani)
        else signal(vzajed);
    cekej(prvni_cekani);
    prvni_citac := prvni_citac - 1;
    druhy_citac := druhy_citac +1;
    if prvni_citac > 0
        then signal(prvni_cekani)
        else signal(druhe_cekani);
    cekej(druhe_cekani);
    druhy_citac := druhy_citac - 1;
end;
S;
if prvni_citac > 0
    then signal(prvni_cekani)
        else if druhy_citac > 0
            then signal(druhe_cekani)
            else signal(vzajed);
```

Výpis 10 - Implementace kritické oblasti

Když proces opouští kritickou sekci, mohl změnit nějakou hodnotu v podmínce *B* a umožnit tak jinému procesu ve vstupu do jeho kritické sekce. Proto musíme sledovat procesy čekající na semaforech *prvni_cekani* a *druhe_cekani* (v tomto pořadí) a u každého testovat jeho podmínku *B*. Může se během tohoto testování stát, že nově ověřená hodnota podmínky je *true*. V tom případě proces vstupuje do své kritické sekce. Jinak musí proces nadále čekat na

semaforech *prvni_cekani* a *druhe_cekani*. Pro každou sdílenou proměnnou *x* může být tedy příkaz

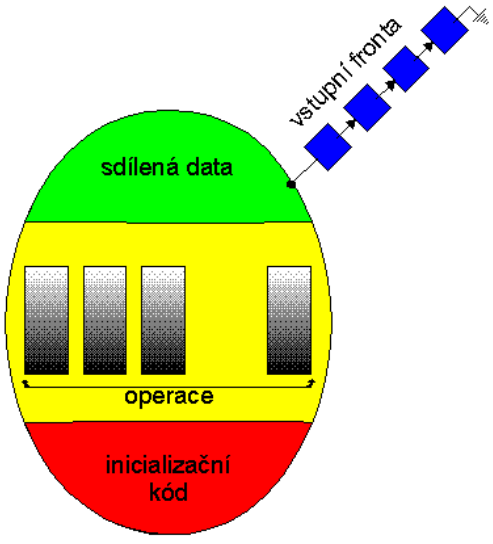
```
region x when B do S;
```

implementován tak, jak ukazuje výpis 10. Připomeňme, že tato implementace vyžaduje aktualizaci hodnoty proměnné *B* vždy, když nějaký proces opouští svou kritickou sekci. Jestliže některé procesy příliš dlouho čekají na nastavení své podmínky do hodnoty *true*, pak neustálé vyhodnocování jejich podmínky může vést ke zpomalení systému. Existují různé optimalizační metody, které lze užít k řešení tohoto problému.

8. MONITORY

Další synchronizační konstrukci vyšší úrovně jsou *monitory*. Monitor je charakterizován množinou programově definovaných operátorů. Reprezentace typu monitor obsahuje deklarace proměnných, jejichž hodnoty definuje procedura či funkce. Syntaxe monitoru je:

```
type jmeno_monitoru = monitor
    deklarace promennych
    procedure entry P1 (...);
        begin ... end;
    procedure entry P2 (...);
        begin ... end;
              .
              .
              .
    procedure entry Pn (...);
        begin ... end;
    begin
        inicializacni kod
    end.
```



Obrázek 2 - Schematický pohled na monitor

Reprezentace typu monitor nemůže být užita přímo jinými procesy. Takže procedura definovaná uvnitř monitoru může využívat pouze proměnných deklarovaných lokálně uvnitř téhož monitoru a formálních parametru. Stejně tak lokální proměnné monitoru mohou využívat pouze jeho lokální procedury.

Konstrukce monitoru zajišťuje, že pouze jeden proces může být v daném čase na monitoru aktivní, takže programátor se nemusí zabývat synchronizačním kódem. Je patrné, že definice monitoru je v mnoha ohledech podobná kritické oblasti a stejně jako existují podmíněné kritické oblasti, tak pro monitor je třeba definovat podmíněnost. Ta se zavádí pomocí typu *podminka*. Programátor, který potřebuje napsat vlastní synchronizační schéma „šité na míru“, může užít proměnných typu *podminka*:

```
var x,y : podminka;
```

Jediné operace, které mohou pracovat s podmínkovými proměnnými jsou *cekej* a *signal*. Operace

```
x.cekej;
```

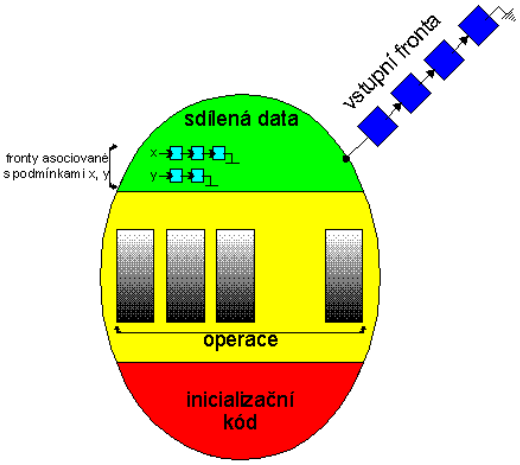
znamená, že proces, který spustil tento příkaz je pozastaven do té doby, dokud jiný proces nezavolá příkaz

```
x.signal;
```

Operace *x.signal* obnovuje vždy právě jeden pozastavený proces. Jestliže žádný proces pozastavený není, nemá tato operace žádný efekt, tzn., že stav *x* je stejný, jako by operace *signal* nebyla spuštěna. To je rozdíl mezi operací *signal* na *podmince* a operaci *signal* na *semaforu*, kde funkce *signal* mění stav vždy.

Nyní předpokládejme, že když proces *P* zavolá funkci *x.signal*, je pozastaven proces *Q* asociovaný také s podmínkou *x*. Je zřejmé, že jestliže pozastavený proces *Q* může dokončit svou činnost, proces *P* musí čekat. Jinak řečeno, oba procesy *P* i *Q* budou současně aktivní uvnitř monitoru. Poznamenejme, že oba procesy mohou pokračovat ve vykonávání svých příkazů, nastala-li jedna za dvou možností:

- *P* čeká, až *Q* opustí monitor, nebo čeká na jinou podmínku.
- *Q* čeká, až *P* opustí monitor, nebo čeká na jinou podmínku.



Obrázek 3 - Monitor s podminkovými proměnnými

Ukažme tuto koncepci při řešení problému hladových filozofů. Řešení nepřipouští zablokování filozofa. Připomeňme, že filozof smí uchopit svou hůlku jen tehdy, má-li přístupné obě dvě. Pro řešení problému musíme rozlišovat mezi třemi stavy, ve kterých se může filozof nacházet. K tomuto účelu definujme následující datovou strukturu:

```
var stav: array (0..4) of (premysli,hladuje,ji);
```

I-tý filozof může nastavit proměnnou stav(i) = ji pouze tehdy, jestliže jeho oba sousedé nejsou ve stavu *ji* tj. *stav*(i + 4 **mod** 5) A *ji* **and** *stav*(i + 1 **mod** 5) A *ji*. Musíme tedy deklarovat

```
var ja: array (0..4) of podminka;
```

kde *i*-ty filozof může čekat má-li hlad, ale nemůže získat obě hůlky, které potřebuje.

```
type dining-philosophers = monitor
    var stav: array (0..4) of (premysli,hladuje,ji);
        ja: array (0..4) of podminka;

    procedure entry seber(i: 0..4);
    begin
        stav(i) := hladuje;
        test(i);
        if stav(i) A ji then ja(i).cekej;
    end;

    procedure entry poloz(i: 0..4);
    begin
        stav(i) := premysli;
        test(i + 4 mod 5);
        test(i + 1 mod 5);
    end;

    procedure entry test(k: 0..4);
    begin
        if stav(k + 4 mod 5) <> ji
            and stav(k) = hladuje
            and stav(k + 1 mod 5) <> ji
        then begin
             stav(k) := ji;
             ja(k).signal;
        end;
    end;

    begin
        for i := 0 to 4 do stav(i) := premysli;
    end;
```

Výpis 11 - Řešení problému hladových filozofů pomocí monitoru

Nyní jsme v situaci, kdy si můžeme popsat řešení problému. Rozdělování hůlek je řízeno monitorem z výpisu 11. *I*-ty filozof musí zavolat operace *seber* a *poloz* instance *dp*, což je instance monitoru *dining-philosophers* v následujícím pořadí:

```
dp.seber(i)
   ...
konzumace
   ...
dp.poloz(i);
```

Je velmi jednoduché ukázat, že žádní dva sousedé nemohou konzumovat současně, takže nemůže dojít k zablokování (deadlocku). Připomeňme, že ale může dojít k umoření filozofa k smrti.

Ukažme si nyní implementaci monitoru pomocí semaforu. Každému monitoru definujme semafor *vzajed* iniciovaný do 1. Proces musí spustit *cekej*(*vzajed*) před tím, než vstoupí do monitoru a musí spustit *signal*(*vzajed*) potom, co monitor opustí.

Další semafor *dalsi* iniciovaný do 0 bude uchovávat procesy pozastavené čekáním na monitor. Celočíselná proměnná *citac* bude uchovávat počet procesů čekajících na semaforu *dalsi*. Potom procedura *P* uvnitř monitoru může být nahrazena následující sekvencí příkazů:

```
cekej(vzajed);
  ...
telo procedury P
  ...
if citac > 0
     then signal(dalsi)
     else signal(vzajed);
```

čímž je vzájemná jednoznačnost uvnitř monitoru zajištěna.

Ukažme nyní implementaci podmínkových proměnných. Pro každou podmínku *x* definujme semafor *x-sem* a celočíselnou proměnnou *x-citac*, obě iniciované do 0. Operace *x.cekej* může být potom implementována tak, jak ukazuje následující výpis.

```
x-citac := x-citac + 1;
if citac > 0
     then signal(dalsi)
     else signal(vzajed);
cekej(x-sem);
x-citac := x-citac - 1;
```

Operace *x.signal* může být potom implementována následovně:

```
if x-citac > 0
then begin
     citac := citac + 1;
     signal(x-sem);
     cekej(dalsi);
     citac := citac - 1;
end;
```

Vraťme se k problému obnovení pozastaveného procesu. Jestliže jsou nějaké procesy pozastaveny na podmínce *x* a některým procesem je spuštěna operace *x.signal*, nastává problém jak určit, který z pozastavených procesů obnovit. Jednoduché řešení tohoto problému je užít FCFS plánování, tj. proces, který čeká nejdéle, bude obnoven jako první. Existuje však mnoho okolností, pro které je tato jednoduchá metoda plánování nepostačující. Proto je definice podmíněného čekání doplněna následovně:

```
x.cekej(c);
```

kde *c* je celočíselná hodnota, která určuje prioritu na podmínce. Tato hodnota je potom uchována spolu s identifikátorem procesu a v okamžiku spuštění funkce *x.signal* je obnoven proces, jehož hodnota priority je nejmenší.

Pro ilustraci tohoto nového mechanismu uvažujme monitor z Výpisu, který kontroluje alokaci jednoho zdroje mezi kooperujícími procesy. Každý proces, který vyžádá alokaci tohoto zdroje, specifikuje maximální dobu, po kterou hodlá zdroj využívat. Monitor potom přidělí zdroj procesu, který ho požaduje na nejkratší dobu.

Proces, který potřebuje užít přidělovaný zdroj, musí ve svém zdrojovém textu obsahovat následující sekvenci:

```
A.ziskat(cas);
   ...
vyuziti zdroje
   ...
A.uvolneni;
```

kde *A* je instance typu *Alokace_zdroje.*

```
type alokace_zdroje = monitor
    varobsazen : boolean;
```

```
x : podminka;

procedure entry ziskat(cas : integer);
begin
    if obsazen then x.cekej(cas);
    obsazen := true;
end;

procedure entry uvolnit;
begin
    obsazen := false;
    x.signal;
end;

begin
    obsazen := false;
end.
```

Výpis 12 - Monitor alokování jednoho zdroje

Bohužel, řešení pomoci monitoru nemůže garantovat, že prioritní přístup ke zdroji bude vždy zajištěn. Speciálně:

- Proces by nemusel nikdy uvolnit zdroj, jakmile by mu byl jednou přidělen.
- Proces by mohl o tentýž zdroj požádat dvakrát (aniž by ho poprvé uvolnil).
- Proces se může pokusit uvolnit zdroj, který mu nebyl přidělen.
- Proces může přistoupit ke zdroji, aniž by zohledňoval jakákoli přístupová práva jiných.

Poznamenejme, že tytéž potíže mohou nastat obecně při řešení problému kritické sekce a jsou bytostně podobné těm problémům, které si vyžádaly vznik řešení pomoci kritické oblasti nebo monitoru. Nejdříve jsme byli znepokojeni nekorektním užitím semaforu a nyní nás trápí chybné použití vyšších programově definovaných operací, které nám překladač nemůže pomoci odhalit.

K zajištění toho, že procesy budou zachovávat patřičné pořadí, musíme kontrolovat všechny programy užívající monitor *alokace_zdroje*. Jsou dvě podmínky, které musíme zajistit, aby systém fungoval korektně:

- Uživatelské procesy musí vždy provádět volání monitoru ve správném pořadí.
- Musíme zajistit odmítnutí nekooperujících procesů, které ignorují vzájemně jednoznačný přístup ke zdroji zajišťovaný monitorem a pokoušející se o přímý přístup ke sdílenému zdroji.

Pouze v případě, že tyto dvě podmínky jsou splněny, je možno garantovat, že nedojde k žádným chybám v časování a že plánovací algoritmus nebude porušen.

Tyto synchronizační postupy mohou být využity u malých a jednoduchých systémů, ne však pro rozsáhlé dynamické systémy. Problém kontroly přístupu ke zdrojům v takových systémech může být řešen pouze pomocí dalších mechanismů, o kterých se zmíníme v kapitole o zabezpečení operačních systémů.