

Od HW k paralelní aplikaci

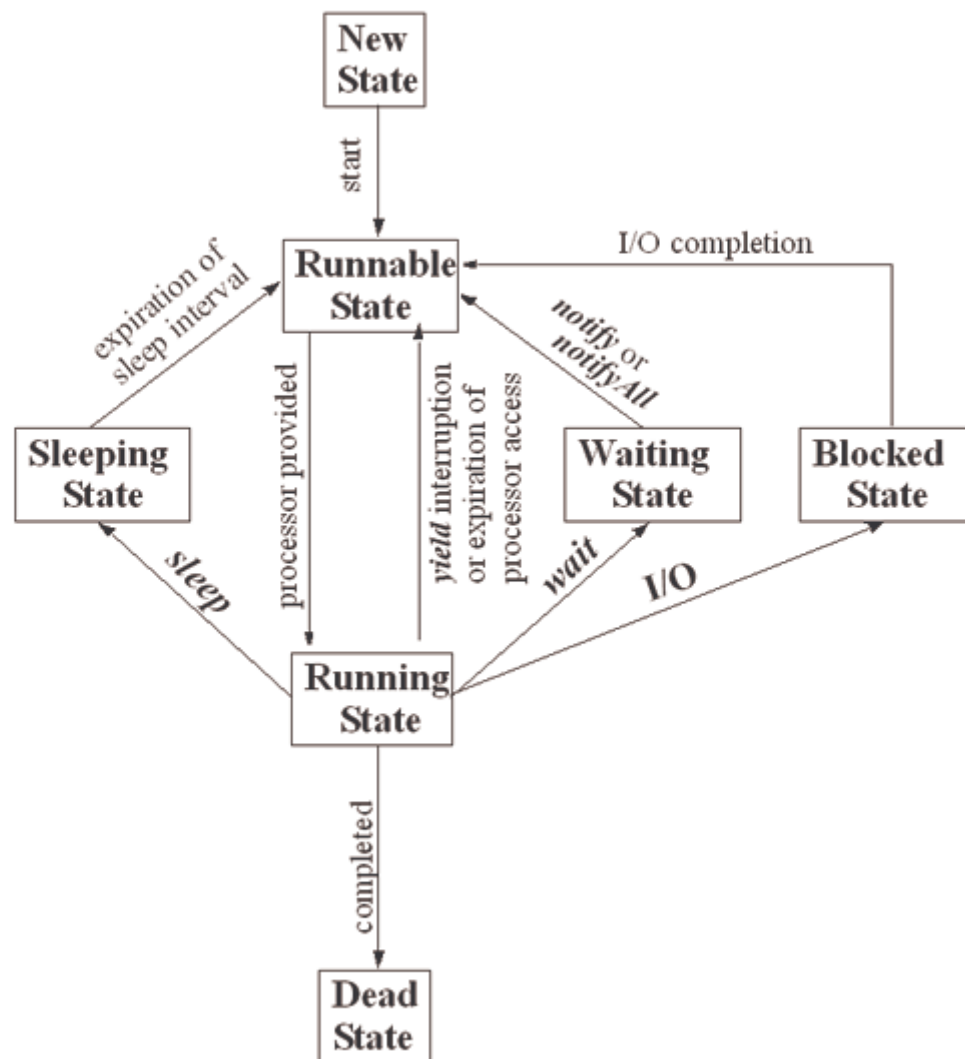
Paralelní aplikace	Paralelní aplikace
	Mezivrstva (Java, ADA, PVM...)
(emulovaný) OS KIV/ZOS, KIV/OS	
(virtualizovaný) HW	

- KIV/PPR
 - Jak vytvořit paralelní aplikaci využívající prostředky poskytovanými OS
 - Mezivrstvy
 - Jak vytvořit paralelní aplikaci využívající prostředky poskytovanými mezivrstvou
 - A trocha z tvorby OS na ukázkou
- MPMD
 - V této přednášce aplikace na jednom počítači se sdílenou pamětí
 - m vláken běží na n procesorech
 - každé vlákno může běžet podle jiného programu

Paralelní program

- Flow Correct
 - chová se deterministicky
 - končí v konečném čase (nejde-li např. o službu)
 - při každém běhu dá stejný výsledek
- Logically Correct
 - Dá správný výsledek

Stavy vlákna



<http://ocw.mit.edu/NR/rdonlyres/Civil-and-Environmental-Engineering/1-124JFall2000/BC616879-7A2D-4A17-B7ED-EC8FFDE1B054/0/threadStates.gif>

- Ukončení vlákna se zde uvažuje samotným vláknem

Zásady psaní paralelních programů

- Minimalizovat počet interakcí vláken
- Minimalizovat dobu trvání interakcí
- Obě uvedené zásady mohou někdy vést ke kompromisu

- Násilné nezasahování do stavu jiných vláken
 - Proměnné
 - Stav, kontext
 - Programátor OS/mezivrstvy bude mít v tomto případě jiný názor

- Rychlostní nezávislost vláken – tj. neuvažovat v návrhu, že dané vlákno poběží nějakou rychlostí
 - V případě hard-real time systémů by se mohla najít výjimka
 - Nespoléhat se na předpokládaný postup plánovače vláken
 - S další verzí se může změnit
 - Může nastat podmínka, která změní plánování, ale vy o ní nevíte
 - Nezneužívat priority

- Interakce musí být atomické
- Navrhnout a dodržet společnou politiku přístupu ke kritickým sekcím
 - Např. dvoufázové zamykání
- Uvědomit si, že přístup k HW (a jiným prostředkům OS/mezivrstvy) je vstup do kritické sekce na úrovni OS/mezivrstvy

Formy interakce

- Primitivní
 - Synchronizace
 - Sdílení dat
 - Zasílání zpráv
- Strukturované
 - Monitor
 - Rendez-vous

Synchronizace

- Semafor
 - KIV/ZOS
 - Atributy
 - Čítač – kolik vláken ještě může vstoupit
 - Fronta čekajících vláken
 - Operace
 - P, Acquire, Enter, Wait – blokující
 - V, Release, Exit, Leave – neblokující
- Bariéra
 - KIV/ZOS
 - Atributy
 - Kolik vláken se má synchronizovat
 - Čítač – kolik vláken už čeká
 - Fronta čekajících vláken
 - Operace
 - Enter

Sdílení dat

- Kritická sekce
 - Řeší se pomocí semaforů
 - P pro vstup
 - V pro opuštění

```
TallestTower.Enter;  
try  
    RescuePrincess;  
    //The dragon and prince Charming  
    //compete for the same resource.  
finally  
    TallestTower.Leave;  
end;
```

- Mutex
 - Pokud v kritické sekci může být jenom jedno vlákno
 - Binární semafor
 - Mutual Exclusion
- Možnost uvíznutí

A.Enter	B.Enter
B.Enter	A.Enter
...	...
B.Leave	A.Leave
A.Leave	B.Leave

- Zámek
 - HW podporovaný mutex
 - Instrukce, které testují hodnotu a při splnění podmínky zapíší jinou hodnotu
 - cmpxchg8b
 - Tzv. Test'n'Set instrukce
 - Atomicky prováděné operace s daty
 - lock inc eax
 - 1 sběrnice do sdílené paměti, ale několik procesorů => zajištění exkluzivního přístupu k sběrnici (tj. paměti)
 - Operace
 - Lock, Unlock
 - Lze ho použít i v jádře OS
 - Pro SMP mají smysl implementace s aktivním čekáním – tzv. spin lock; jinak verze s yield
 - Na jednoprocessorovém systému to nemá smysl
 - Hodí se pro případy, kdy se čeká, že vstup do kritické sekce bude poskytnut relativně brzy

```
mov edx, DWORD(-1) //-1 zamčeno
//otestujeme stav zámku, 0 = odemčeno
spin: mov eax, [lockState]
      test eax, eax
      jnz spin

//zkusíme ho zamknout s -1
lock cmpxchg [lockState], edx

//nepředběhl nás jiný procesor?
//původní lockState je v eax
test eax, eax
jnz spin
```

- Reference Counter

- Určité operace jsou „umzačeny“ (neprovedou se) dokud má reference counter vyšší hodnotu než je stanovený limit (obvykle nula)
- Bitmapa se nevykresluje na desktop, dokud se mění

```
Control.Canvas.BeginUpdate;  
try  
    DoSomePainting(Control.Canvas);  
finally  
    Control.Canvas.EndUpdate;  
end;
```

- Objekt se neuvolní z paměti, dokud ho někdo zná

```
//implementace IUnknown  
ULONG AddRef() {  
    return InterlockedIncrement(&refc);  
    //viz HW podporovaný mutex  
}  
  
ULONG Release() {  
    ULONG rc = InterlockedDecrement(&refc);  
    if (!rc) delete this;  
    return rc;  
}
```

- Knihovna se nenačte 2x, ale pouze 1x

```
Lib1:=LoadLibrary(libName);  
Lib2:=LoadLibrary(libName);  
FreeLibrary(lib1);  
    //knihovna je stále  
    //v adresovém prostoru  
FreeLibrary(lib2);
```

- Zasílání zpráv
 - Rozdělení viz předchozí přednáška
 - Hodí se pro situace, kdy není nutné reagovat ihned
 - Operace
 - Send, SendMessage, PostMessage
 - Receive, GetMessage, PeekMessage
 - Triviální implementace spočívá ve vložení zprávy do bufferu příjemce pro *send*
 - A v testování bufferu, vybrání zprávy z bufferu příjemcem pro *receive*

```
private FIFO Strings;
```

```
synchronized public void PostMsg(String msg) {  
    Strings.Push(msg); //neblokující, volá "Sender"  
}  
synchronized public String ReceiveMsg {  
    return Strings.Pop; //blokující "while empty"  
}
```

- Později viz jako příklad monitoru
- Shatter Attack
 - V dřívějších verzích (bez patche) než Windows Vista mohl každý proces zaslat zprávu komukoliv
 - Mohly tak spolu komunikovat dva procesy, každý s jiným oprávněním
 - Šlo injektovat vlastní kód do procesu s vyšším oprávněním a tam ho spustit (SetWindowsHookEx)
- => komunikace se řídí protokolem a ten může mít i jiné, než výkonnostní slabiny/nedostatky

- Monitor

- Pasivní objekt, který poskytuje služby aktivním vláknům
- Vlákna se navzájem neznají, znají pouze monitory
- Monitor zajišťuje konzistenci dat implementací vzájemného vyloučení
- Má
 - Stav daný stavem proměnných
 - Frontu čekajících vláken
- Pokud by monitor strážil např. integer velikosti registru, pak zapisovat ho může jenom jedno vlákno, číst ho může více vláken
 - Aneb optimalizace monitoru, která ho zredukovala na HW podporovaný zámek
- Pokud monitor stráží proměnou ADT, na 32bitovém stroji je to i 64bitový integer, pak v něm může být aktivní pouze jedno vlákno
- Lze naprogramovat monitor, který může strážit i více nezávislých proměnných a tedy v něm může být aktivních několik vláken
 - Špatná dekompozice programu
- Dříve se používala implementace podle Hoareho (spolu s Hansenem stáli u zrodu monitorů)
 - Vlákno čeká na vstupní podmínce, P, a vstoupí do monitoru jakmile předchozí signalizuje opuštění monitoru, V
 - Zbytečně komplikovaná implementace
 - Navíc, plánovače už nejsou kooperativní
- Dnes Mesa – čekající se pouze uvede do runnable

monitor.V

thread = monitor.PopWaitingThread

thread.Resume //pozor, oba mohou sdílet CPU

thread.yieldTo //tak přímo to nešlo ani s Win16

▪ V kterém moderním OS to dnes uděláte? ;-)

```
class MailBox {                                // monitor's class

    // attributes
    private StringBuffer message;
    private boolean empty;

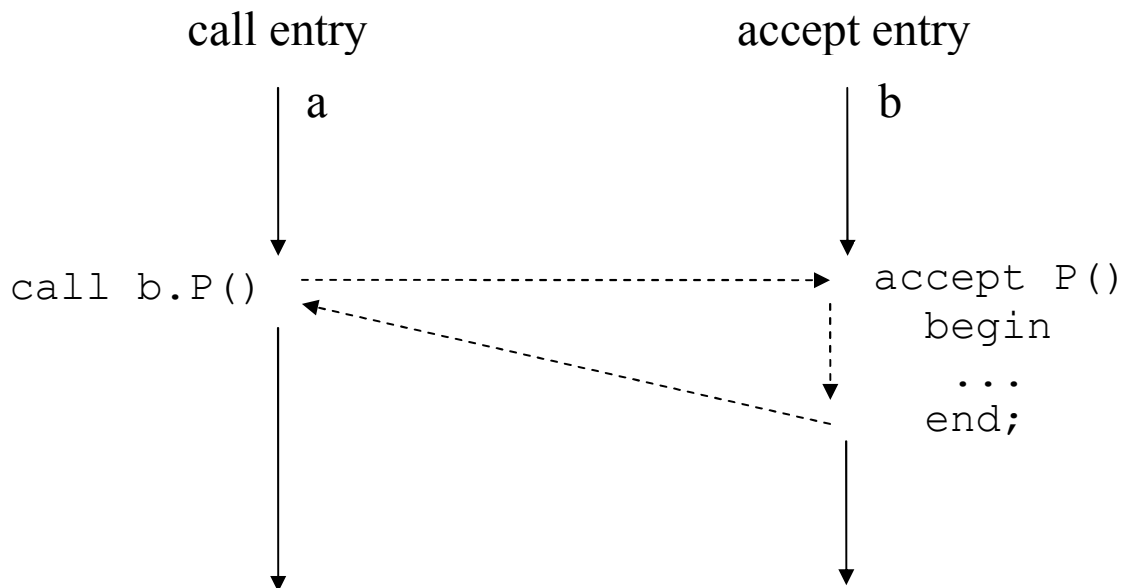
    // methods
    public MailBox ()
        {empty = true;}                        // monitor's constructor

    synchronized public void
        write (StringBuffer input) {
        while (!empty) try {
            wait ();
        } catch (InterruptedException e) {}
        message = input;    // !!! it writes
                             // only a reference

        empty = false;
        notify();
    }

    synchronized public StringBuffer read () {
        while (empty) try {
            wait ();
        } catch (InterruptedException e) {}
        empty = true;
        notify();
        return message;
    }
}
```

- Rendez-Vous



- Synchronní – call a accept jsou blokující
- Propojí se adresní prostory obou procesů
- /vláken – vždyť paměť je přece psaná na proces?!
 - Každé vlákno má svůj zásobník a adresy aktuálních lokálních proměnných funkcí se liší podle aktuálního vnoření volání funkcí
- Sdružuje se synchronizace a výměna dat
- Náročná implementace v distribuovaném prostředí
- Accept entry je atomická operace se vzájemným vyloučením
 - Tj. může být aktivní pouze jeden – volající
 - Tj. s r-v lze implementovat monitor

```

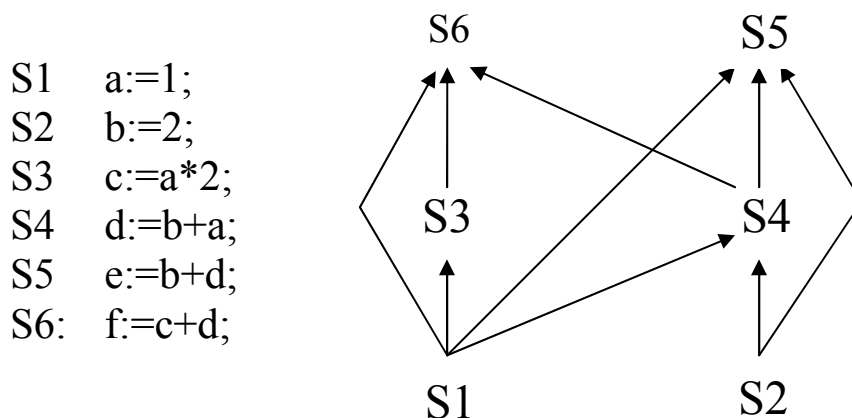
select
    accept P1 ()
or accept P2 ()
...
or accept Pn ()
end;

```

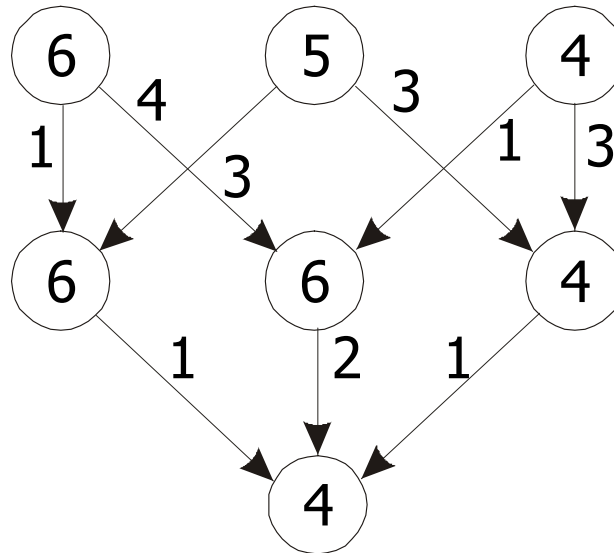
- r-v nejsou jenom samá pozitiva
 - Ada (bude později)
 - Všechny Tasky jsou aktivní, paralelně běžící vlákna
 - Monitor soupeří o zdroje s „normálním“ vláknem
 - Více přepínání kontextu, což je drahé
 - Minimálně se vynutí dvě přepnutí při každém zavolání (a opuštění) r-v

Modely paralelních výpočtů

- Analýza uvíznutí
- Analýza výkonnosti
- Důkaz správnosti chování
- Precedenční graf
 - Vyjadřuje návaznost činností
 - Acyklický graf
 - smyčka znamená uvíznutí

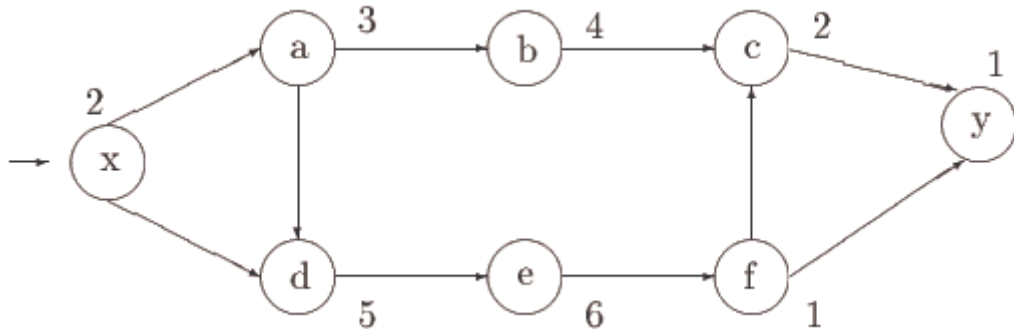


upravené <http://www.csc.villanova.edu/~japaridz/Archive/1300/lect7.1/sld006.htm>

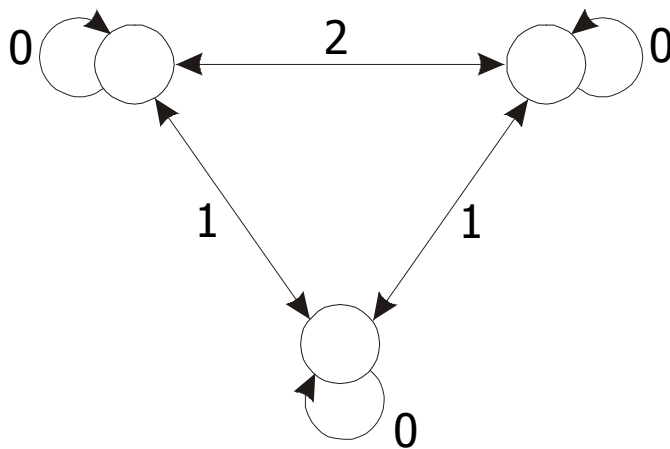


- Uzel je ohodnocen náklady na výpočet – např. doba výpočtu
- Hrana je ohodnocena náklady na komunikaci
 - Má smysl u distribuovaných výpočtů
 - U sdílené paměti je můžeme zanedbat
- Umožňuje analyzovat urychlení výpočtu
 - Doba výpočtu na jednoprocessorovém systému je dána součtem dob všech výpočtů
 - Doba výpočtu na víceprocesorovém systému je nejdelší cesta grafem, který reprezentuje alokaci výpočtů na jednotlivé procesory
 - V uvedeném obrázku je
 - Výpočet na 3 procesorech: 16
 - Výpočet na 1 procesoru: 35
 - Urychlení: $35/16 = 2,1875$
 - Nelze tedy tvrdit, že s n procesory něco poběží n -krát rychleji
- Synchronizace znamená, že se čeká na dokončení všech činností, z jejichž uzlů vedou hrany do daného uzlu

- Alternativě můžeme uvažovat grafy, kdy
 - Uzel představuje synchronizační bod
 - Hrany jsou náklady na výpočet
 - Hodí se do prostředí se sdílenou pamětí

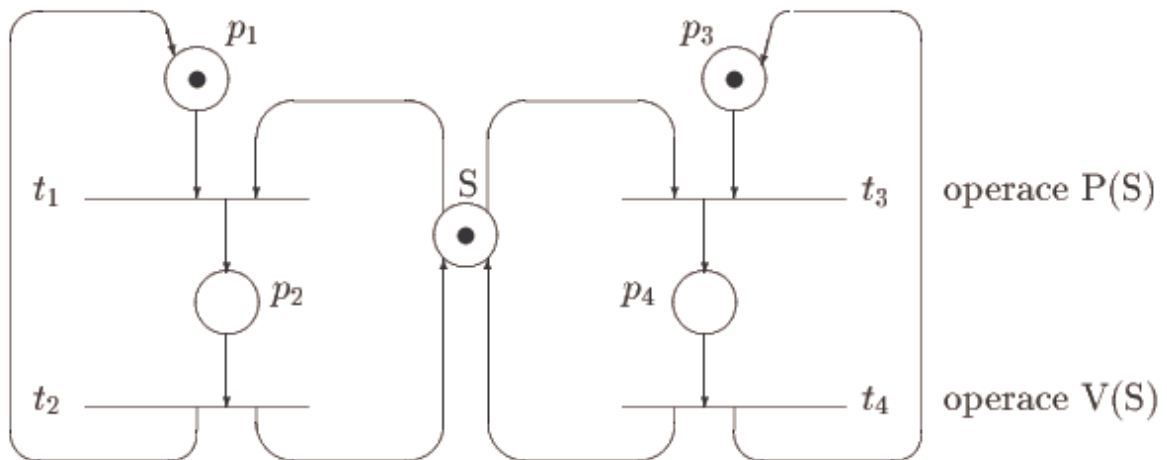


- Komunikační model systému



- Popis systému
- Hrany jsou ohodnoceny náklady na komunikaci mezi jeho uzly
- Uzly můžeme ohodnotit výkonem

- Petriho síť



Vzájemné vyloučení, skripta KIV/PPR

- Dva typy uzlů
 - Místo – kruh, modeluje činnost
 - Přechod – úsečka, modeluje událost
- Hrana udává přechod
- Stav systému vyjadřuje značí
 - Plné kruhy
 - Celočíselná hodnota
 - Počet plných kruhů v uzlu
- K přechodu dojde tehdy, jestliže má každá vstupní hrana značení
- Počet značek může reprezentovat počet zdrojů
 - Např. producent-konzument
- p_2 a p_4 na obrázku jsou kritické sekce dvou procesů/vláken, které přistupují ke sdíleným datům
- existuje-li takový stav, kdy není možný žádný přechod, pak v systém může nastat uvíznutí
- přechod je okamžitý, s časem počítá stochastická Petriho síť

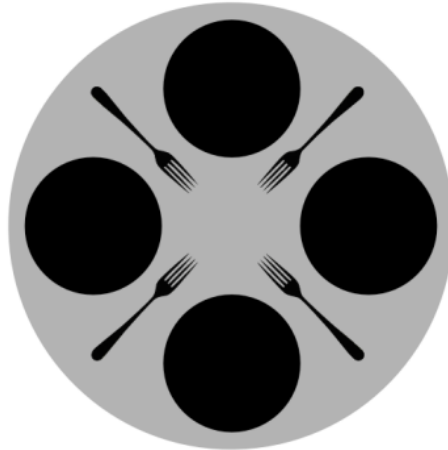
- Důkaz správnosti programu
 - Proměnné popisují stavy a vlastnosti programu
 - Každá proměnná $x \in \mathcal{V}$
 - \mathcal{V} (ocabulary) je množina všech proměnných
 - Aplikací logických operátorů na proměnné a konstanty se zkonstruují výrazy, které popisují nějakou vlastnost programu
 - $=, >, \&, \text{not}, \forall, \exists, \dots$
 - Jazyk k popisu programu lze rozšířit o vlastní operátory – např. časové
 - $p \mathcal{W} q$ p čeká (pokud není) q
 - Jde o automat s množinou proměnných \mathcal{V} , počátečním stavem θ a konečnou množinou přechodů \mathcal{T}
 - Co hledáme je tzv. invariant
 - Výraz, který říká, že např. v kritické sekci vzájemného vyloučení bude vždy maximálně jenom jeden proces
 - Ověřující podmínka efektu přechodu mezi dvěma stavy říká, že každý následník t stavu φ je stav ψ
 - $\{\varphi\} \vdash \{\psi\}$
 - Základní invariance

$$\frac{\begin{array}{l} B_1 \quad \theta \rightarrow \varphi \\ B_2 \quad \{\varphi\} \mathcal{T} \{\varphi\} \end{array}}{\varphi}$$

- Monotónnost invariance

$$\frac{p, p \rightarrow q}{q}$$

- Večeřící filozofové



- Čtyři talíře, čtyři vidličky, čtyři filozofové
- Filozof buď přemýšlí, nebo jí (špagety, které nabírá ze středu stolu)
- K jídlu potřebuje dvě vidličky, může si vzít pouze tu vidličku, která je bezprostředně u něj
- V okamžiku, kdy všichni najednou vezmou svou levou vidličku, nastane deadlock
- M filozofů, M vidliček
 - Nejdříve levou vidličku, pak pravou
 - $\varphi: \neg(at_f[j] \wedge at_f[j \oplus_M 1])$
pro $\forall j \in [1..M]$
 - Sice symetrické řešení (levá, pak pravá) s identickými komponentami (snadnější úloha), ale je možný deadlock

- “Rebel Inside, Rebel for Life”
 - Jeden z filozofů bude mít jinou vnitřní logiku
 - Bude si brát vidličky v opačném pořadí než ostatní
 - Jenomže už nemáme všechny komponenty identické

- Ostraha jídelny
 - Jídelnu bude hlídat semafor, který bude inicializován na hodnotu $<1, M-1>$
 - Komponenty jsou identické, ale přibyl nám do programu další prvek

- Ohodnocení vidliček (zdrojů) – priorita
 - Filozof si vždy vezme nejprve vidličku s nejvyšší prioritou
 - Všechny komponenty jsou identické, stále stejný počet prvků v programu

const

BufferSize = somePositiveInteger;

var full:TSemaphor(0);

empty:TSemaphor(BufferSize);

buffer:**array**[0..BufferSize-1]**of** TBufferElement;**thread** producer;**var** index:integer = 0;

item:TBufferElement;

begin

item:=Produce;

empty.P;

buffer[index]:=item;

full.V;

index:=++index % BufferSize;

end;**thread** consumer;**var** index:integer = 0;

item:TBufferElement;

begin

full.P;

item:=buffer[index];

empty.V;

index:=++index % BufferSize;

Consume(item);

end;

- Jak to bude vypadat, když bude několik producentů/konzumentů?
- Jak to bude vypadat s použitím monitorů/rendez-vous?