# Od HW k paralelní aplikaci

<table>
<tr><td rowspan="2">Paralelní aplikace</td><td>Paralelní aplikace</td></tr>
<tr><td>Mezivrstva (Java, ADA, PVM…)</td></tr>
<tr><td colspan="2">(emulovaný) OS<br>KIV/ZOS, KIV/OS</td></tr>
<tr><td colspan="2">(virtualizovaný) HW</td></tr>
</table>

- 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 ukázku

- 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
  - Pozor – řeklo se „správný výsledek“, ne že správný výsledek bude pořád stejný

# Nedeterminismus logicky správného výsledku

- Mějme model nějakého efektu, a hledejme paralelně parametry tohoto modelu tak, abychom dosáhli co nejmenší průměrné odchylky vypočítaných hodnot od naměřených hodnot
  - Tj. jde o redukční operaci

- Může se nám tedy stát, že dvě různá vlákna nám mohou vygenerovat dvě různé množiny parametrů, pro které vypočítají dvě různé množiny odchylek
  - 1; 2; 1; 2; 1
  - 1; 0; 3; 2; 1

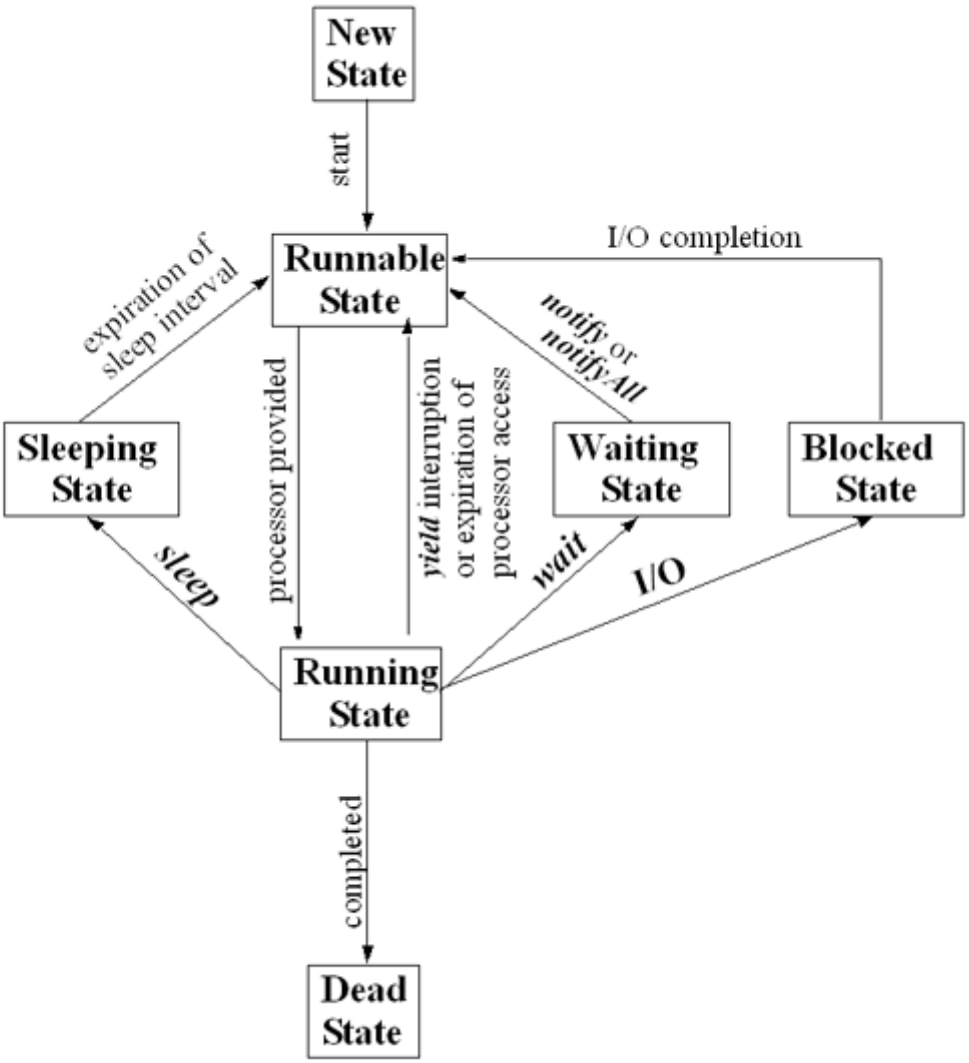
  - Pokud bychom počítali průměrnou odchylku, tak nám v obou případech vyjde stejné číslo
  - Ale které z nich z použijeme a které je správně?
    - To záleží na tom, které vlákno odevzdá své výsledky ke globálnímu porovnání jako první
    - Zvolená metrika vhodnosti výsledku nám to neumožňuje lépe rozhodnout

- Kdybychom vyhodnocovali sériový program, pak se nám bez této znalosti může mylně zdát, že na rozdíl od paralelního programu pracuje správně, protože dá vždy stejný výsledek
  - Přičemž výsledek sériového programu nemusí být lepší než některý z jiných výsledků paralelního výpočtu

- Některé knihovny, např. Intel Threading Building Blocks, nabízejí deterministickou verzi redukční operace
  - Není tak rychlá jako nedeterministická verze
  - Ale měla by vždy dát stejný výsledek, protože se snaží práci pokaždé rozdělit stejně
  - Tj. neřeší za nás metriku vhodnosti řešení!

- Další komplikací je fakt, že na FPU sčítání ani násobení nejsou nezbytně asociativní, ačkoliv jsou komutativní
  - Tj. (a +b) + c nemusí dát ten samý výsledek jako a + (b + c)
  - Důvodem je konečný počet číslic reprezentujících daná čísla

# 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 vyjímka
  - 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 //spinlock
  - 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 [rax]
      - 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 jednoprocesorovém 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
```

- Zámek s úsporou energie
  - Intel® 64 and IA-32 Architectures Optimization Reference Manual 2011

  - Neoptimální verze

```
while(!acquire_lock())
  { Sleep( 0 ); }
do_work();
release_lock();
```

    - Volání sleep(0) přepíná do kontextu jádra
      - => pomalé, počet cyklů zpoždění znásobí, třeba 10000 cyklů?
    - Pokud to nesoupeří s jiným vláknem, procesor by byl vytížen na 100% a nemohl by přejít do úsporného režimu,jak by mohl s instrukcí pause
  - Power-Consumption Friendly
    - Využívá instrukci pause – úsporný režim

```
if (!acquire_lock()){
  /* Spin on pause max_spin_count times before
     backing off to sleep */

  for(int j = 0; j < max_spin_count; ++j) /
    /* intrinsic for PAUSE instruction*/
    _mm_pause();
    if (read_volatile_lock()) {
      if (acquire_lock()) goto PROTECTED_CODE;
     }
   }

  /* Pause loop didn't work, sleep now */
  Sleep(0);
  goto ATTEMPT_AGAIN;
}
PROTECTED_CODE:
  do_work();
  release_lock();
```

- Spinlock je de-facto změna hodnoty integeru a datový typ bool je také interně jenom integer o dvou možných hodnotách

```
class SpinLock {
private:
    std::atomic<bool> mLock(false);
        // holds true when locked
 public:
    void lock(){
     while(!mLock.compare_exchange_strong
                                (false, true));
    }

    void unlock() { lock = false;   }
};
```

- Takže se dá rovnou použít tzv. flag, který ovšem st8le skrývá výše uvedenou instrukci cmpxchg (a na kterou směřuje otázka u zkoušky)

```
class SpinLock {
private:
    std::atomic_flag mLock = ATOMIC_FLAG_INIT;

public:
    void lock(){
        while(mLock.test_and_set
                    (std::memory_order_acquire));
    }

    void unlock()  {
        mLock.clear(std::memory_order_release);
    }
};
```

- Reference Counter
  - Určité operace jsou „uzamčeny“ (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

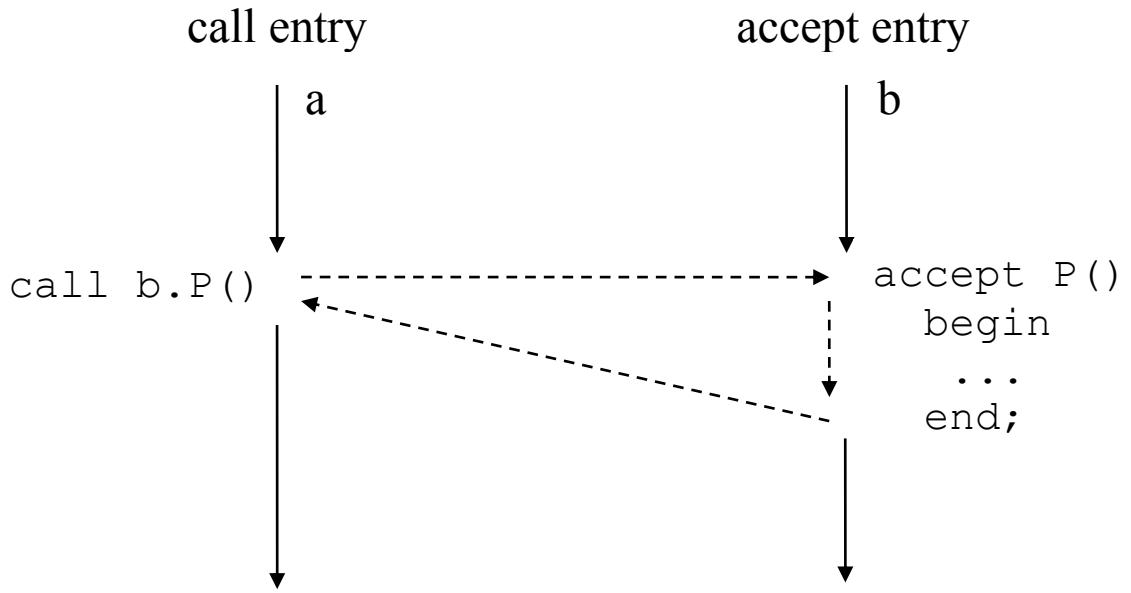
- Monitor
  - Pasivní objekt, který poskytuje služby aktivním vláknům
  - Vlákna se navzájem neznají, znají pouze monitory
  - Monitor zajišťuje konzistenci dat implementací vzájemného vyloučení
  - Má
    - Stav daný stavem proměnných
    - Frontu čekajících vláken
  - Pokud by monitor strážil např. 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  //oba thready mohou plánovány na
stejný CPU a plánovač může mít jiný plán než vy
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
  - 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;
```

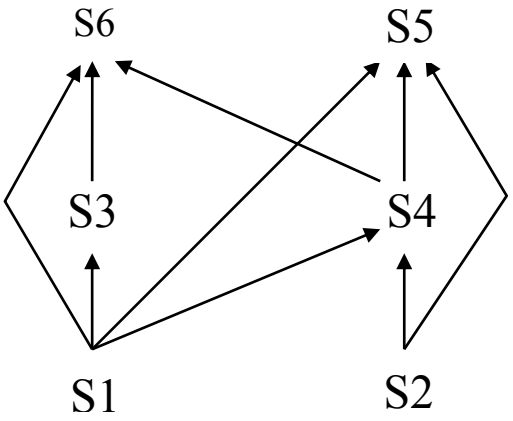
- r-v nejsou jenom samá pozitiva
  - Ada (bude později)
    - Všechny Tasky jsou aktivní, paralelně běžící vlákna
    - Monitor soupeří o zdroje s „normálním“ vláknem
    - Více přepínání kontextu, což je drahé
    - Minimálně se vynutí dvě přepnutí při každém zavolání (a opuštění) r-v

# Modely paralelních výpočtů

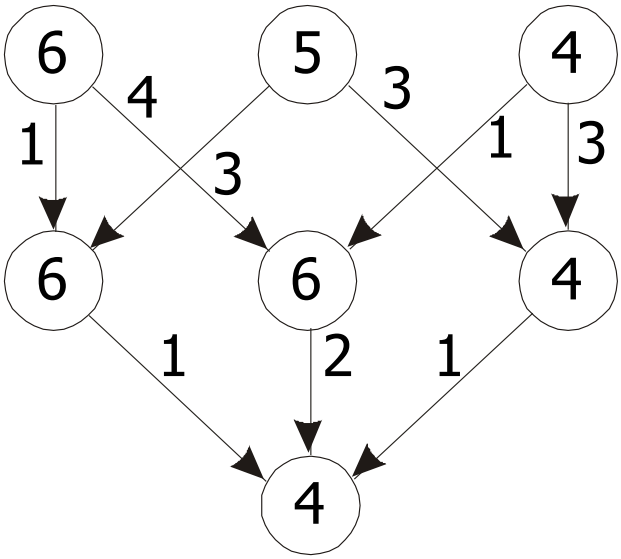
- Analýza uvíznutí
- Analýza výkonnosti
- Důkaz správnosti chování

- Precedenční graf
  - Vyjadřuje návaznost činností
  - Acyklický graf
    - smyčka znamená uvíznutí

```
S1   a:=1;
S2   b:=2;
S3   c:=a*2;
S4   d:=b+a;
S5   e:=b+d;
S6:  f:=c+d;
```

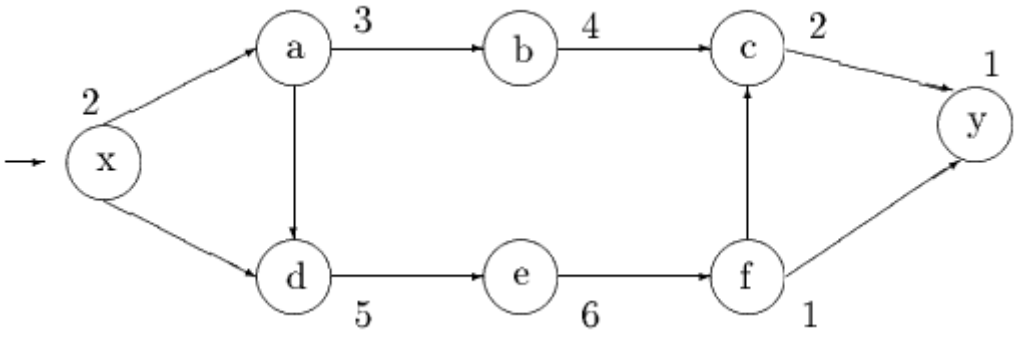



upravené http://www.csc.villanova.edu/~japaridz/Archive/1300/lect7.1/sld006.htm

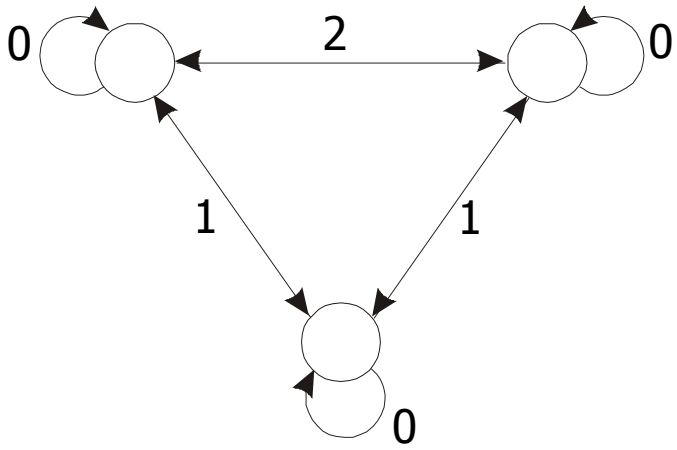



- Uzel je ohodnocen náklady na výpočet – např. doba výpočtu
- Hrana je ohodnocena náklady na komunikaci
  - Má smysl u distribuovaných výpočtů
  - U sdílené paměti je můžeme zanedbat
- Umožňuje analyzovat urychlení výpočtu
  - Doba výpočtu na jednoprocesorovém systému je dána součtem dob všech výpočtů
  - 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 automaticky 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
  - Hrana jsou náklady na výpočet
  - Hodí se do prostředí se sdílenou pamětí

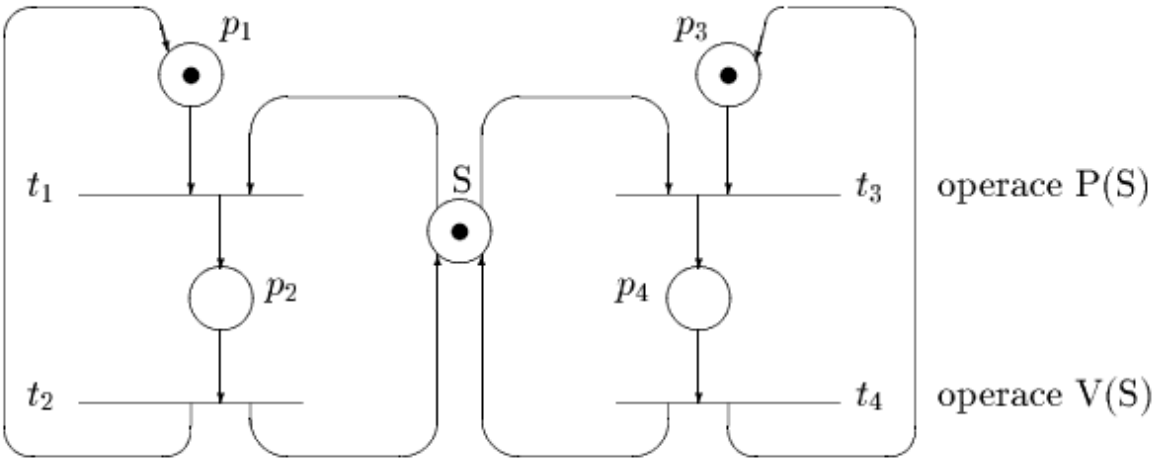



- Komunikační model systému




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

- Petriho sítě




Vzájemné vyloučení, skripta KIV/PPR

  - Dva typy uzlů
    - Místo – kruh, modeluje činnost
    - Přechod – úsečka, modeluje událost
  - Hrana udává přechod
  - Stav systému vyjadřuje znační
    - 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

  - p2 a p4 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

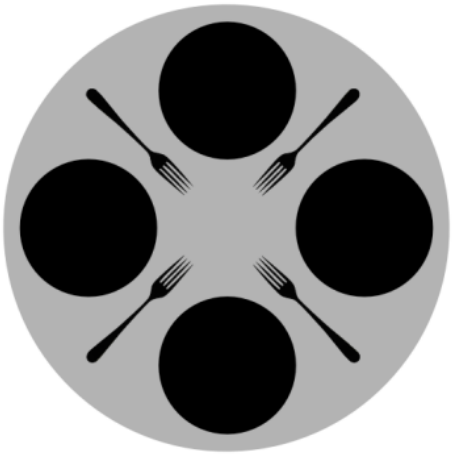
- Důkaz správnosti programu
  - Proměnné popisují stavy a vlastnosti programu
    - Každá proměnná $x \in \mathcal{V}$
    - $\mathcal{V}$(ocabulary) je množina všech proměnných
  - Aplikací logických operátorů na proměnné a konstanty se zkonstruují výrazy, které popisují nějakou vlastnost programu
    - =, >, &, not, ∀, ∃, …
    - 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 $\theta$ 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
      - $\{\varphi\}\ t\ \{\psi\}$
    - Základní invariance

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

- 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

- Aloha, CSMA/CD, …
  - Filozof se pokusí vzít si obě vidličky postupně
  - Když to nevyjde, uvolní, co si vzal, a počká náhodnou dobu, než to zkusí znovu
  - Sice to lze těžko formálně zaručit, ale v praxi to funguje u Ethernetu do překročení hraniční hodnoty aktivity vysílačů

```
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?