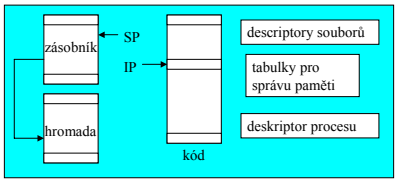
Týden 6
Programování se sdílenými proměnnými
(Shared Variables, SV)

Možnosti programování se sdílenými proměnnými

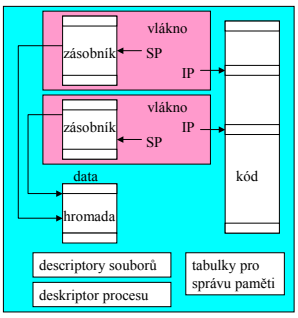
1. Sekvenční jazyk + paralelizující kompilátor = zatím málo efektivní (implicitní paralelismus)
2. Sekvenční jazyk + knihovní programy (nízká úroveň) (programování s vlákny Pthreads, Solaris, Java,...)
3. Paralelní programovací jazyky (např. SGI Power C, C//, ...)
4. Sekvenční jazyk + **OpenMP** API, standard nezávislý na platformě: - direktivy pro kompilátor (pragma),
   - i knihovní programy.
5. Zasílání zpráv (MPI, PVM) a datově paralelní programování (HPF) lze použít i na systémech se sdílenou pamětí (SMP).

Procesy



Procesy „těžké váhy“ - úplně separátní programy s vlastními proměnnými, zásobníkem a alokací v paměti (adresovým prostorem). O.K. pro OS.
Vytvoření procesu v Unixu může trvat stovky tisíc taktů CPU! Vytvoření vláken je mnohem rychlejší.

Procesy a vlákna



- Vlákno = IP + SP, myid, priorita, registry, privátní paměť.
- Proces startuje jako základní vlákno.
- Vlákno může vytvořit další vlákna za pomoci kernelu nebo uživatelské knihovny.
- Proces tak může mít mnoho vláken; všechna sdílejí týž adres. prostor.
- Vlákna jsou lehká = málo alokované paměti, málo kopírování, menší režie.

Proč používat vlákna?

- **Pro reprezentaci souběžných entit (systémové programování a uživatelská rozhraní)**
  Komerční Thread Libraries
  Win32 threads (Windows NT and Windows 95).
  Pthreads (Portable Operating System Interface, POSIX Thread Interface).
  (SGI IRIX, Sun Solaris, HP-UX, IBM AIX, Linux, atd.).
  Solaris threads (SunOS 5.x).
- **Multiprocessing kvůli výkonnosti**
  - Souběžnost je pod kontrolou programátora.
  - Programy by mohly být napsány sekvenčně.
  - Toto multivláknové programování je snadnější.

Příklad: Win32 Threads

Thread Management: `CreateThread(), SuspendThread(), ResumeThread(), ExitThread(), TerminateThread(), WaitForSingleObject(), WaitForMultipleObjects(), CloseHandle()`

Critical Sections: `InitializeCriticalSection(), DeleteCriticalSection(), EnterCriticalSection(), LeaveCriticalSection()`

Mutexes: `CreateMutex(), OpenMutex(), WaitForSingleObject(), WaitForMultipleObjects(), ReleaseMutex(), CloseHandle()`

Semaphores: `CreateSemaphore(), WaitForSingleObject(), WaitForMultipleObjects() ReleaseSemaphore(), CloseHandle()`

Events: `CreateEvent(), SetEvent(), ResetEvent(), PulseEvent(), WaitForSingleObject(), WaitForMultipleObjects(), CloseHandle()`

Interlocked Operations: `InterlockedIncrement(), InterlockedDecrement(), InterlockedExchange()`

Thread Priorities: `SetThreadPriority(), GetThreadPriority()`

Model zpracování vláken: klíčové body

- Soubor procesorů, soubor vláken.
- Vlákna mezi sebou jsou na stejné úrovni.
- Dynamické vytváření vláken (fork - join, thread_create, ...)
- Může být mnohem víc vláken než procesorů.
- Vlákna se dynamicky přepínají mezi procesory.
- Vlákna sdílejí přístup do paměti.
- Vlákna je třeba synchronizovat.
- Podpora procesů, vláken, přepínání kontextu, zámků, bariér: HW, SW (multiprocesorový kernel, knihovny).
- Volání systému nebo knihovní programy se nazývají bezpečné k vláknům (*thread-safe*), když mohou být volány z více vláken současně a výsledky jsou vždy správné.

7

Úloha synchronizace a komunikace

- "Paralelní počítač je kolekce procesorů, které kooperují a komunikují, aby řešily velké problémy rychle."
- Každá komunikace přes sdílenou paměť s privátními cache generuje výpadek a vyžaduje synchronizaci;
- časová režie = (výpadek zápisu) + doba zápisu + synchronizační režie + výpadek čtení + doba čtení
- umělá komunikace vlivem
  - omezené kapacity pamětí cache
  - falešného sdílení (první procesor píše, druhý čte z téhož bloku, každý užívá jinou adresu)

Podmínky pro paralelní čtení a zápisy ve společném adresovém prostoru (AP)

I_k = množina paměťových míst čtená procesem P_k
O_h = množina paměťových míst modifikovaná procesem P_h

(Bernsteinovy) postačující podmínky:

$$I_1 \cap O_2 = \varphi$$
$$I_2 \cap O_1 = \varphi$$
$$O_1 \cap O_2 = \varphi$$

Příklad: Procesy id = 0,1,..., P-1 nesmí obsahovat příkazy typu

- `sum = sum + myPart [id]` # → jedině v kritické sekci !

```
# pragma multithreadable
{
    x = n;
    x = m;
}
```

```
# pragma multithreadable
{
    a = b;
    c = a;
}
```

9

Komunikace p2p se synchronizací pomocí obyčejných proměnných

p1	p2
a = 3.1415;	
flag = 1;	**while (flag==1) ;**
	print a ;
a = b;	
inc(count,1);	**check(count,1);**
	c = a;

```
# pragma multithreadable
{
    { a = 3.1415;  set(flag); }
    {check(flag); print a; }
}
```

```
# pragma multithreadable
{
    { a = b;  inc(count,1); }
    {check(count,1); c = a; }
}
```

- Zápis do A a do flagu nelze přehodit
- Čtení A a flagu nelze přehodit

→ moderní procesory ale často pořadí přístupů do paměti mění ! Flag a count musí být proto označeny (deklarovány) jako synchronizační proměnné.

Příklad špatné synchronizace

p1	p2
lock (L)	
a = 1;	
b = 3.1415	
unlock (L)	**lock (L)**
	... = a;
	... = b;
	unlock (L)

Muselo by být jasné, že p1 zamkne lock první ! Jinak nebude fungovat.

```
# pragma multithreadable {    # → direktiva dělej multivláknově:
    { acquire(lock); a = b; release(lock); }
    { acquire(lock); c = a; release(lock); }
}
```

Skupinová komunikace AAS přes SM

p1	p2	p3
od1k2 = 1.12;	**od2k1 = 2.23;**	**od3k1 = 0.34;**
od1k3 = 3.1415;	**od2k3 = 4.15;**	**od3k2 = 1.67;**
bar (aas, 3)	**bar (aas, 3)**	**bar (aas, 3)**
… = od2k1;	**… = od1k2;**	**… = od1k3;**
… = od3k1;	**… = od3k2;**	**… = od2k3;**

- Zápisy musí dokončit před bar
- Čtení musí začít po bar

Synchronizace přístupů ke sdílené proměnné

Paralelní zpracování si lze představit jako prolnutí atomických akcí jednotlivých procesů. Při interakci procesů ne všechna prolnutí jsou přijatelná. Synchronizace má zabránit nežádoucím prolnutím:

```
Thread 1:                    Thread 2:
    ...                      ...
EnterCS;                     EnterCS;
sum = sum + f(x);            sum = sum + f(y);
LeaveCS;                     LeaveCS;
...                          ...
```

~~LD1~~	LD1
~~ADD1~~	ADD1
~~LD2~~	ST1
~~ADD2~~	LD2
~~ST1~~	ADD2
~~ST2~~	ST2

Zde místo CS stačí atomické FA (Fetch&Add)

Synchronizační mechanizmy

0. **Atomicita** znamená spojení atomických akcí jednotlivých instrukcí v nějaké sekvenci do jedné atomické sekvence (transakce). Umožňuje paralelismus (paralelní transakce). Nejčastěji se realizuje vzájemným vyloučením, ale není to jediný nebo nejlepší způsob.
1. **Vzájemné vyloučení** znamená, že v určitém okamžiku může být v kritické sekci (CS) programu pouze jeden proces → CS se provádí atomicky a vynucuje si sekvenční zpracování. Nejčastěji se implementuje pomocí binární sdílené proměnné zámku (též mutex od mutual exclusion).
2. **Podmínková sychronizace**. Jeden proces nastaví boolovskou podmínku, další proces(y) ji testují a čekají na její nastavení. (= zpoždění akce až nastane podmínka). Sem patří bariéra, events (int), flags (bool).

Vlastnosti vzájemného vyloučení

0. V daném okamžiku provádí CS nejvýš jeden proces.
1. Je vyloučen deadlock, z procesů čekajících na vstup do CS musí být nakonec jeden úspěšný.
2. Žádný proces nevyhladoví, dostane se na každého.
3. Proces nesmí v CS skončit, být přerušen nebo vyřazen ze zpracování.
4. Chce-li vstoupit do CS jen jeden proces, nebude zbytečně zdržován.

Nástroje pro řešení vzájemného vyloučení

1. Holý stroj, jen atomické čtení a zápis do sdílené paměti: Dekkerův algoritmus pro 2 procesy (3 sdílené proměnné). pro N procesů: pořadové lístky.
2. HW podpora: atomické instrukce test&set, fetch&inc, swap nebo dvojice instrukcí LL,SC (Load Locked, Store Conditional). Sdílená proměnná lock = 0 (nikdo není v CS), jinak 1 (obsazeno). Procesy cyklicky testují proměnnou lock ve smyčce (spinning), CPU zaměstnána (busy-waiting). Zámek může uvolnit jen proces, který jej užívá.
3. Semafory: vyšší úroveň, nestrukturované. Nezáporné int s manipulované atomickými operacemi P a V. Kernel udržuje frontu procesů čekajících na semafor a přiděluje jej spravedlivě. Odpadá busy-waiting, nicméně zůstává režie kernelu. Binární nebo obecný semafor, ovládání z libovolného procesu.
4. Monitory: více strukturované než semafory.

Kritická sekce s pořadovými lístky

```
int ticket = 1, next = 1;   /*sdílené proměnné*/
process CS[i = 1 to n] {
  int myturn;
  while (true) {
    myturn = FA(ticket,1);
    while (myturn != next) skip;
    critical section;
    next++; /*nemusí atomicky*/
    noncritical section;
  }
}
```

Deklarace procesů, obdoba PLACED PAR i = 1 FOR n SEQ | CS

Nebo zpoždění úměrné rozdílu myturn - next

Algoritmus s pořadovými lístky
Sdílené proměnné ticket, next.
Zajišťuje spravedlivý (FIFO) přístup k zámku.

Kritická sekce se zámkem "test&set" (TS)

```
bool lock = false;     /*shared lock */
process CS[i = 1 to n] {
  while (true) {
    while (TS(lock)) skip;
    critical section;
    lock = false;
    noncritical section;
  }
}
```

Pokud zámek obsazený, (lock = 1), čekej

Nevýhoda: špatná výkonnost, TS chce soustavně zapisovat do bloku paměti cache, kopie téhož bloku v ostatních cache musí být proto pokaždé zneplatněny nebo aktualizovány. Soupeření o paměť, proměnná zámku je „horké místo".

Kritická sekce se zámkem “test a test&set”

```
bool lock = false;        /*shared lock */
process CS[i = 1 to n] {
  while (true) {
    while (lock) skip;    /* cekej na lock=0 */
    while (TS(lock)) {    /* pokud je 0, nastav 1*/
      while (lock) skip;  /*jinak znovu cekej na 0*/
    }
    critical section;
    lock = false;
    noncritical section;
  }
}
```

Testování v lokálních cache, soupeření o sběrnici odpadá.

Po uvolnění zámku několik nebo všechny procesory zase budou chtít čerstvou kopii a budou soupeřit o sběrnici.

Podmínková synchronizace

Návěští (flags) signalizují splnění nějaké synchronizační podmínky. Jako návěští lze použít normální proměnné a cyklicky je testovat (busy-waiting, CPU zaměstnána).

Kvůli velká časové režii cyklického testování globálních proměnných se zavádějí podmínkové proměnné (též events) a operace nad nimi:

wait (cond_var) -- čekej na splnění podmínky
signal (cond_var) -- signalizuj splnění podmínky
signal_all (cond_var) -- rozhlaš splnění podmínky všem čekajícím procesům
status (cond_var) -- vrací počet procesů čekajících na splnění podmínky

Příkaz wait nezaměstnává CPU, frontu procesů (vláken) čekajících na cond_var udržuje systém.

Interakce procesů se sdíleným AP

Jde o komunikaci spojenou se synchronizací. Analogie ke skupinovým komunikacím u zasílání zpráv. Spíše pro OS.

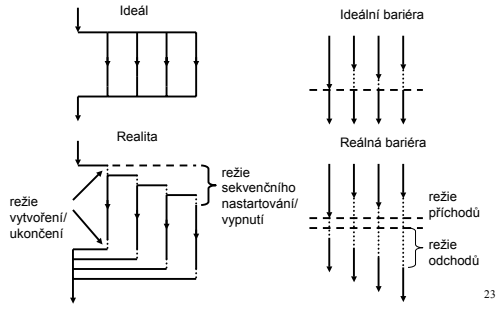
1. Asynchronní komunikace s jednomístným buferem (jednou sdílenou proměnnou): producent - konzument, pipeline.
2. Několik producentů, několik konzumentů, mnohomístný (kruhový) buffer (sdílená fronta).
3. Producenti specifickým konzumentům (křížový přepínač).
4. Čtenáři a písaři sdílené datové struktury (seznamu, tabulky, souboru, databáze apod.)
   - písař má exkluzivní přístup do sdílených dat
   - když žádný písař sdílená data nepoužívá, čtenáři je mohou souběžně číst (exkluzivní přístup: buď písař nebo čtenáři).
5. Stolující filozofové - procesy žádají současný přístup více než k jednomu prostředku.

Otázky výkonnosti multivláknování

- Režie spojená s udržováním vláken.
- Vyrovnání zátěže.
- Granularita (velikost úseků kódu do synchronizace)
- Soupeření o paměť.
- Algoritmus úlohy.

– Režie vytváření a rušení vláken.
– Režie plánování vláken.
– Režie synchronizace vláken.

Režie vytvoření, rušení a synchronizace vláken



Opakované řešení n úloh: par nebo bar?

par i bar čekají až dokončí všech n úloh:

```
while (true) {
  par [i=1 to n]
    kód úlohy i
  rap
}
```

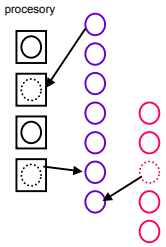
```
proces Worker[i=1 to n]
  while (true) {
    kód úlohy i
  bar(n)
  }
}
```

Krajně neefektivní, je mohem dražší vytvářet a rušit procesy než je synchronizovat !

Bariérová synchronizace

Režie plánování vláken

- Vlákna mojou být buď *připravená (běžet)* nebo být *pozastavená*.
- Zásady plánování:
  - vlákna připravená k běhu nahrazují pozastavená;
  - připravená vlákna vysoké priority předbíhají připravená vlákna nízké priority
  - stojící připravená vlákna nahrazují běžící vlákna (preemption)
- Přepnutí vlákna stojí jistý čas.
- Nové naplnění cache stojí čas.

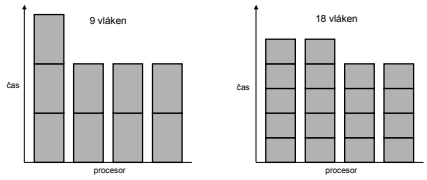


Multivláknová výkonnost a granularita

- Granularita = měřítko objemu výpočtů mezi operacemi s vlákny a synchronizačními operacemi.
- Jemná granularita = málo výpočtů mezi operacemi
- Hrubá granularita = mnoho výpočtů mezi operacemi
- Je třeba zajistit rovnováhu mezi:
  - příliš jemnou granularitou = příliš mnoho režie s vlákny.
  - příliš hrubou granularitou = špatné rozdělení zátěže.

Vyrovnávání zátěže více vláken

- Vyrovnávání zátěže = všem procesorům stále přisunovat práci.
- Velký počet malých, stejně velkých vláken znamená lepší vyrovnání zátěže.
- Zvol dostatek vláken, aby se dala dobře vyrovnat zátěž.

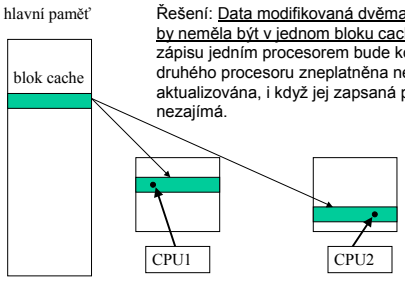


Multivláknová výkonnost: soupeření o paměť

- Výpadky cache jsou velmi drahé - mohou způsobit soupeření o paměť.
- Přepište program aby se zvýšila lokalita přístupů do paměti.
- Optimalizace sekvenčního chování cache bude minimalizovat soupeření o paměť při multivláknování.
- Toto je velká výhoda metodologie vývoje multivláknových programů jako sekvenčních.
- Pozor na past "falešného sdílení".

Falešné sdílení

Řešení: Data modifikovaná dvěma procesy by neměla být v jednom bloku cache. Při zápisu jedním procesorem bude kopie bloku druhého procesoru zneplatněna nebo aktualizována, i když jej zapsaná položka nezajímá.



Multivláknová výkonnost: použitý algoritmus

- Sekvenční složka α % omezuje zrychlení na $S = 1/\alpha$ (Amdahlův zákon).
- Multivláknový algoritmus může být méně efektivní než nejlepší sekvenční algoritmus.
- Problém rozkladu může zvýšit celkovou zátěž (např. algoritmus PSRS).
- Problém rozkladu může snížit celkovou zátěž (např. při diskrétní optimalizaci).
- Problém rozkladu změní chování cache - často k lepšímu.