Západočeská univerzita
FAKULTA APLIKOVANÝCH VĚD

Okruhy otázek ke státní závěrečné zkoušce z předmětu
Systémové programování (SP)

Operační systémy (OS)
Paralelní programování (PPR)
Formální jazyky a překladače (FJP)
Výkonnost a spolehlivost čísl. systémů (VSP)

Studijní program:	3902	Inženýrská informatika
Obor:	2612T025	Informatika a výpočetní technika – Softwarové inženýrství
	3902T031	Softwarové inženýrství

Akademický rok: 2005/2006

Obsah:

1 Paralelní výpočetní prostředí se sdílenou a distribuovanou pamětí - charakteristika

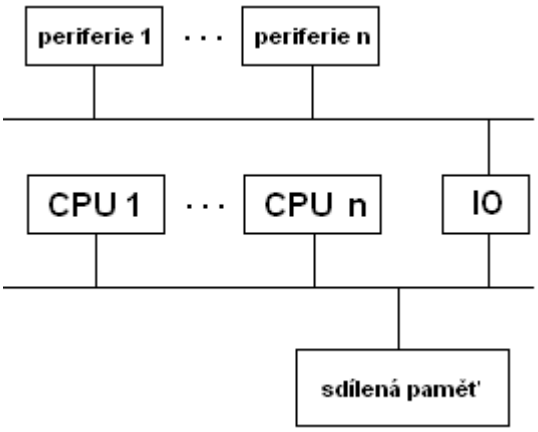
1.1 Multiprocesory se sdílenou pamětí

- V paralelním výpočetním prostředí se sdílenou pamětí jsou procesory připojeny přes paralelní sběrnici ke společné paměti. Sběrnice (spojovací subsystém) je kritickým místem této architektury a na jeho propustnosti silně závisí počet procesorů, které lze do systému připojit
- Multiprocesory se sdílenou pamětí můžeme rozdělit na:
    - **Homogenní** – Výpočetní procesory jsou stejného typu a mají stejnou množinu instrukcí.
    - **Nehomogenní** – Procesory jsou různého typu.

- Homogenní multiprocesory můžeme dále dělit na:
    - **Symetrické** – Všechny procesory v systému mají stejné možnosti práce (např. přístup ke sdíleným periferním prostředkům), operační systém zachází se všemi stejně.
    - **Asymetrické** – Procesory nemají stejné možnosti

1.1.1 Symetrické procesory

- Všechny univerzální procesory mají v systému stejné možnosti využití zdrojů a z hlediska operačního systému jsou rovnocenné.
- Kód i data operačního systému jsou alokovány ve sdílené paměti. Žádný proces tedy není vázán na konkrétní procesor a v průběhu života může vystřídat několik procesorů.

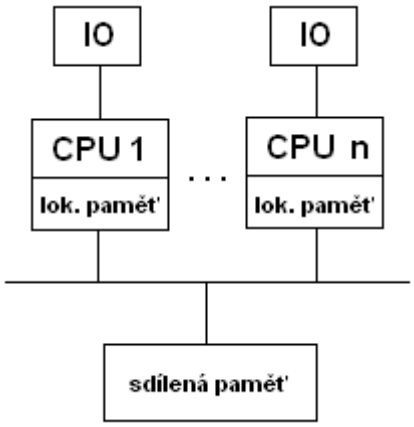


- Výhody symetrické architektury:
    - **Jednoduchost řízení na úrovni OS** – Naprosto stejný kód jádra může být vykonáván paralelně všemi procesory. Jádro operačního systému při většině svých akcí nemusí mezi procesory rozlišovat a může s nimi zacházet anonymně.
    - **Univerzálnost** – Libovolný výpočet dekomponovatelný na paralelní procesy je v této architektuře velmi dobře realizovatelný
    - **Spolehlivost** – V systému je N – násobná redundance procesorů, při výpadku jednoho zpravidla stačí restartovat právě běžící proces.
- Nevýhody symetrické architektury
    - **Výkonnost** - Centralizované uložení všech informací ve sdílené paměti způsobuje velké zatížení spojovacího subsystému (spojuje procesory s pamětí a

dalšími prvky architektury) a z toho plyne degradace výkonnosti pro velký počet procesorů.

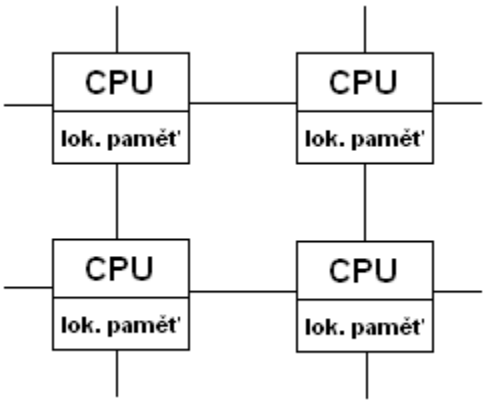
1.1.2 Asymetrické procesory

- Všechny procesory jsou stejně jako v předchozím případě stejného typu.
- Každý však kromě sdílené využívá i vlastní (lokální) paměť a může mít realizované vazby na vlastní periferní zařízení. Z hlediska operačního systému už nelze považovat tyto procesory za rovnocenné.
- Programový kód OS může být společný pro všechny procesory a může být uložen ve sdílené paměti, popřípadě distribuován mezi sdílenou paměť a lokální paměti procesorů.
- Asymetrie je způsobena různými funkčními schopnostmi procesorů. Operační systém nemůže alokovat kterýkoliv proces na kterýkoliv procesor, musí brát v úvahu požadavky procesu na zdroje systému.
- Výhodou je především možnost snížení zatížení spojovacího subsystému vhodným rozvržením procesů do lokálních pamětí. Ve sdílené paměti je nutné ponechat pouze data nutná pro komunikaci mezi procesy.
- Nevýhodou je pak složité řízení na úrovni operačního systému, OS musí rozlišovat procesory a je např. složitý přenos dat mezi periferním zařízením připojeným k procesoru 1 a zařízením připojeným k procesoru 2.



1.2 Multiprocesory s distribuovanou pamětí

- Základní prvky této architektury jsou procesory s vlastní lokální pamětí vázané mezi sebou rychlými sériovými linkami.
- Ve struktuře není přítomná žádná sdílená paměť a lokální paměti obsahují programový kód a data pro proces alokovaný na příslušném prvku sítě.
- Procesy vzájemně komunikují zasíláním zpráv po sériových linkách. Prvky sítě jsou tak v podstatě kompletní počítače s omezenými IO možnostmi.
- Hlavní výhodou je použitelnost i pro úlohy vyžadující tzv. masivní paralelismus (tj. je zapotřebí stovky až tisíce procesorů), neboť zde není žádný „úzký profil“ ve formě společného komunikačního subsystému. Navíc může být výrazně omezena režie vytváření procesů a přepínání kontextu (v každém prvku sítě může být jen jeden proces).
- Nevýhodou je hlavně aplikační citlivost (opak univerzálnosti). Konkrétní topologie sítě totiž není univerzálně vhodná pro všechny paralelní algoritmy.



- Architektura obecně:
  - Jednotlivé prvky jsou v podstatě samostatné počítače, tzv. uzly – počet N.
  - Existují mezi nimi komunikační linky, nebo lépe komunikační subsystém. Komunikační kanály tvoří komunikační síť.
  - Existují 3 základní možnosti realizace:
    - Univerzální počítačová síť (např. PVM). SW cestou se vytvoří virtuální multiprocesor, který počítá jednu aplikaci. Vlastnosti silně závisí na propustnosti sítě
    - Paralelní počítač. Přímo určený k paralelním výpočtům, je to hlavní funkce systému. Výpočetní prostředí je homogenní (stejné uzly, SW integrace do jednoho celku). Lze docílit větší urychlení. Např. cluster pracovních stanic (Lyra na KMA). Typický SW je MPI (Message Passing Interface)
    - Embedded systém. Např. počítačový tomograf. Paralelní aplikace je vestavěna ve specifickém HW, programy v ROM, pevné (optimalizované) propojení uzlů, aplikace běží sama. Lze dosáhnout nejvyššího urychlení. Jazyk Occam, transputery.
- Architektura z pohledu HW:
  - Fyzická topologie komunikační sítě může být:
    - pevná – dvojice procesorů jsou spojené
    - flexibilní – komunikační přepínač (switch), uvnitř přepínače je přepínací matice (circuit switching) nebo přenášení zpráv po částech (packet switching)
  - Nejčastější varianty pevné topologie:
    - 2D mřížka
    - Toroid
    - 3D mřížka
    - n rozměrná binární krychle (v každé souřadnici jen dvě možnosti 0 nebo 1), souřadnic je n → celkem $N = n^2$ uzlů, poloha např. 01011. Každý uzel má spojení se všemi ostatními
  - Parametry, kterými se topologie charakterizuje:
    - maximální komunikační vzdálenost (d_{max})
    - počet sousedů
    - snaha je, aby bylo d_{max} co nejmenší a počet sousedů taky (kvůli HW propojení)

2 Základní programové modely pro paralelizaci výpočetní činnosti (MPMD, SPMD, MPSD)

- Vlastnímu programování paralelní úlohy by měla předcházet fáze analýzy, ve které se rozhoduje o základním výpočetním přístupu, který se použije pro dekompozici výpočtu na jednotlivé složky - procesy
- Základní používané modely pro paralelizaci výpočetní činnosti jsou MPMD, SPMD, MPSD

2.1 MPMD (Mulitple Program Multiple Data)

- Jedná se o dekompozici výpočtu na relativně samostatné činnosti, z nichž některé mohou být vykonávány paralelně.
- Tento přístup se využívá pro relativně složitou činnost a málo objemná data (tj. důraz je kladen na dekompozici činnosti, data potřebná pro dílčí činnost k ní pak logicky přiřadíme – označují se jako lokální data procesu; je snaha minimalizovat potřebu globálních dat, která nejsou přiřazena k žádnému konkrétnímu procesu).
- Provedenou dekompozici lze obvykle vyjádřit precedenčním grafem, ve kterém hrany symbolizují činnosti a uzly potřebu synchronizace
- Využitím modelu *MPMD* nemusí být sledováno pouze urychlení výpočtu. V mnoha případech (např. řízení v reálném čase, simulační programy) se tento model využívá s ohledem na lepší strukturování programu.
- Programový model MPMD lze implementovat jak na jednoprocesorovém počítači, tak na multiprocesorech různého typu.
- Lze například vytvořit paralelní program např. v jazyce Ada na jednoprocesorovém počítači, odladit ho a pak přenést beze změny kódu (vše zařídí překladač) na symetrický multiprocesor. Na něm budou jednotlivé procesy probíhat fyzicky paralelně a výpočet bude rychlejší.

2.2 SPMD (Single Program Multiple Data)

- Tento model se používá v případě, kdy relativně jednoduchá činnost je prováděna nad objemnými daty.
- Zpracovávaná množina dat (obvykle jednorozměrné nebo vícerozměrné homogenní pole prvků s jednoduchým typem) se v tomto případě rozdělí na m částí. Vytvoří se k procesů ($k \leq m$) pracujících podle stejného programu a každý zpracuje jednu nebo několik strukturně podobných (ale hodnotami různých) částí dat.
- Je sledováno výhradně výkonnostní hledisko (urychlení výpočtu). Už při analýze se uvažuje fyzicky paralelní výpočet a neuvažuje se, že by program mohl být použit i na jednoprocesorovém stroji (kde by běžel pseudoparalelně a nedošlo by tedy k žádnému urychlení).
- Model *SPMD* se používá především pro násobení matic nebo iterační numerické řešení parciálních diferenciálních rovnic (geometrická dekompozice).
- Průběh paralelizace (většinou cyklu) je následující: Různé iterace se svěří různým vláknům, nemá cenu zakládat větší počet procesů než je k dispozici procesorů (docházelo by k přepínání kontextu a tedy ke zpomalení), každé vlákno realizuje přibližně (počet iterací / počet vláken) iterací

2.3 MPSD (Multiple Program Single Data)

- Jedná se o zřetězené zpracování dat, analogií z běžného života by byla průmyslová pásová výroba
- Jedná se o zpracování rozsáhlého proudu datových prvků, přičemž nad jednotlivými prvky jsou vykonávány nějaké (libovolně složité) operace, které je možné svěřit různým specializovaným procesům a vykonávat je paralelně pro několik prvků datového proudu.

3 Ukazatele paralelizace – složitost, urychlení, účinnost

3.1 Složitost

- Anglicky complexity, worst-case complexity
- Je to funkce *f(n)*, jejíž hodnota je pro konkrétní algoritmus úměrná maximální době jeho výpočtu; maximum se bere přes všechny možné vstupy algoritmu s rozměrem *n* (např. rozměr matice, počet čísel v paralelně realizovaném součtu apod.)
- Např. sekvenční algoritmus pro součet *n* čísel má složitost *f(n) = n*, protože maximální doba výpočtu je úměrná počtu čísel *n*.
- Složitost se označuje písmenem *O*, v předchozím případě tedy *O(n)* znamená, že doba výpočtu je lineárně závislá na počtu čísel *n*.
- Pro paralelní součet prováděný na $p = n / 2$ procesorech je složitost *O(log n)*, kde logaritmus je s libovolným základem

3.2 Urychlení

- Anglicky speedup
- Označuje se písmenem *S*
- Obvykle se vyhodnocuje jako poměr doby výpočtu nejlepšího známého sekvenčního algoritmu a doby výpočtu paralelního algoritmu na témže (paralelním) počítači, využíváme-li *p* procesorů.
- *Anomální urychlení*
    - Při zběžném pohledu se zdá samozřejmým, že urychlení výpočtů nemůže být lepší než lineární podle počtu procesorů, tato úvaha však opomíjí, že více procesorů pohromadě může poskytovat nejen akumulaci výkonu, ale i jiných zdrojů
    - Vedle běžných, vcelku logických, odlehčení např. rozdělení paměťového prostoru na více menších částí může velmi omezit nutnost odkládání na disky, lze někdy pozorovat i urychlení superlineární, tedy lepší než lineární. Nejčastěji se tak děje ve dvou případech:
        - Efekt cache-paměti – Rozdělením výpočtu mezi více procesorů může dojít za příznivých podmínek k daleko častějšímu uplatnění lokálních cache-pamětí. Každý lokální výpočet je pak prováděn rychleji než v případě výpočtu jedním procesorem. Pokud algoritmus sám o sobě vykazoval dobré urychlení, může pak být celkové urychlení lepší než lineární
        - Anomálie při prohledávání – Paralelizované prohledávací algoritmy metodou „uřezávání“ pracují často rychleji, než by odpovídalo lineárnímu urychlení. Je to hlavně díky tomu, že paralelizovaný algoritmus je vlastně odlišný od sekvenčního a umožňuje většinou rychlejší upřesňování průběžného výběrového kritéria.

3.3 Účinnost

- Anglicky efficiency
- Jedná se o urychlení dělené počtem použitých procesorů
- Např. nejlepší sekvenční algoritmus se počítá na uvažovaném počítači 10 s, výpočet hodnoceného paralelního algoritmu pro $p = 4$ procesory trvá 5 s. Urychlení je v tomto případě 2.0 a účinnost je 0.5.

3.4 Amdahlův zákon

- Úzce souvisí s urychlením.
- Kvalitní paralelní algoritmy musí nepochybně vykazovat dostatečné urychlení při dostatečné účinnosti (tj. využití procesorů).
- Dosažené urychlení je omezeno Amdahlovým zákonem, který bere v úvahu, že výpočet zpravidla nelze paralelizovat úplně, určitá část musí být provedena sekvenčně. Označíme-li tuto část *f*, pak pro výpočet probíhající na *p* procesorech je dosažitelné urychlení *S* limitováno podle vzorce:
- $$S \leq \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f}$$
- Na první pohled to tedy vypadá, že i při sebevětším počtu procesorů nikdy nedosáhneme většího urychlení než *1/f*. Amdahlův zákon je však poplatný své době a obavy vychází z ne zcela oprávněných předpokladů:
    - Byl uvažován pouze tradiční způsob postupné paralelizace. Při tomto způsobu jsou nejdříve objevena místa s největšími výpočetními nároky. Ty se pak víceméně tradičními postupy paralelizují. Ostatní místa se nechají provádět sekvenčně a jsou prohlášena za nenapravitelně sekvenční. Jistě, jejich paralelizace by možná vyžadovala větší úsilí, ale nelze ho vyloučit jednou provždy. Navíc pokud nějaká část úlohy musí probíhat sekvenčně, ještě to neznamená, že ostatní procesory musí zahálet.
    - Druhým mnohdy zastřeným předpokladem je neměnnost podílu *1/f*. Použití paralelních počítačů umožňuje s rostoucím počtem procesorů řešení stále rozměrnějších úloh. Podle praxe se přitom zároveň ukazuje, že objem sekvenčních výpočtů narůstá velmi pomalu a podíl *1/f* se tak rychle zmenšuje.

4 Kategorizace programovacích prostředků pro paralelní programování (jazyky a knihovny, příklady)

- Většina programovacích jazyků používaných v současné době v sobě přímo obsahují prvky pro paralelní programování, nebo pro ně existují rozšiřující knihovny
- Programovací prostředky se liší podle toho, zda se jedná o paralelní program pro volně vázané systémy (distribuovaná paměť) nebo o systémy se sdílenou pamětí.
- Dále je třeba rozlišovat, zda se paralelní výpočet dělí na vlákna či procesy.
    - Proces je aktivita probíhající podle svého programu, má vlastní kontext a adresní prostor a může být vnitřně dále členěna na vlákna. Používají se často ve volně vázaných systémech.
    - Vlákno je rovněž aktivita probíhající podle svého programu, vlákna v jednom procesu však sdílí adresní prostor, což může být využito pro komunikaci. Používají se často na jednoprocesorových počítačích a obecně v systémech se sdílenou pamětí

4.1 Programovací prostředky pro prostředí se sdílenou pamětí

- Mezi běžně používané jazyky pro prostředí se sdílenou pamětí patří C s využitím knihovny POSIX, Java či C#. Používá se i jazyk Ada. Všechny tyto jazyky využívají k paralelizaci výpočtu vlákna (anglicky thread)

4.1.1 C a vlákna POSIX

- V standardním Ansi C nejsou primitiva pro vytváření vláken
- Vše potřebné pro práci s vlákny je v knihovně POSIX
- Obsahuje tři základní typy objektů
    - Vlákna – vlastní program vlákna, který je uložen v metodě `void* prog_name(void* arg)`. Vlákno se rozběhne hned po vytvoření
    - Mutexy – zámky používané pro vzájemné vyloučení
    - Podmínkové proměnné – jsou reprezentovány pomocí tzv. handle, což je zobecněný ukazatel. Podmínkové proměnné se používají pro synchronizaci vláken. Implementují frontu, kde vlákna čekají na splnění podmínky, neimplementují test podmínky (ten musím být v kódu vlákna). Aby test podmínky + případná změna proměnné byla atomická akce, je třeba sdružit podmínkovou proměnnou s mutexem.

4.1.2 Java

- V programovacím jazyce Java jsou objekty a metody potřebné pro práci s vlákny dostupné ve standardních knihovnách.
- Základem je třída `Thread`, každá její instance reprezentuje jedno vlákno.
    - Výkonný kód vlákna se nachází v metodě `run()`
    - Vlákno se vytváří standardně konstruktorem a spouští se voláním metody `start()`
- Pro synchronizaci vláken se používají monitory, které jsou součástí každého objektu

4.1.3 Ada

- Programovací jazyk vytvořený DoD, používá se při vytváření softwaru pro řízení vojenských zařízení, jako raket, sond atp.
- Paralelní části výpočtu se označují jako task.
- Mohou být prováděny na jednom procesoru, více procesorech nebo více počítačích, není to však vyjádřeno v kódu programu
- Pro synchronizaci tasků se používá rendezvous.

4.2 Programovací prostředky pro prostředí s distribuovanou pamětí

- Mezi nejčastěji používaná jazyky patří C a Fortran. Pro oba existují knihovny PVM (Parallel Virtual Machine) a MPI (Message Passing Interface). Zajímavý je rovněž jazyk Occam.

4.2.1 Occam2

- Primárně určen jako programovací prostředek pro sítě transputerů (rychlý 32-bitový jednočipový mikropočítač s vlastní pamětí, spojený s okolím několika rychlými sériovými linkami).
- Bývá označován jako assembler pro transputery, svojí koncepcí je ale univerzálním programovacím jazykem nižší úrovně pro volně vázané architektury.
- Komunikace mezi uzly sítě je synchronní (oba účastníci na sebe musí počkat) a anonymní (účastníci se neznají, znají pouze jméno kanálu, po kterém komunikují).
- Komunikační protokol může být jednoduchý a není třeba vyrovnávací paměť.
- Komunikační kanály (logické) jsou v jazyce zavedeny jako speciální datový typ (*chan*). Logické kanály jsou jednosměrné, na rozdíl od fyzických kanálů transputeru, které jsou obousměrné.
- Primitivní procesy
    - Odpovídají základním příkazům v sekvenčním programovacím jazyce
    - Příkaz klasického jazyka je zde chápán jako elementární výpočet schopný paralelní koexistence s jinými procesy
    - Pokud chceme provést více příkazů sekvenčně, musíme to explicitně předepsat pomocí *SEQ*.

4.2.2 PVM

- *PVM* je v nejobecnějším pohledu *univerzální výpočetní model* pro paralelní programování, který má dobrou naději stát se mezinárodním standardem.
- Je implementováno pro paralelní počítače různého typu (tj. z pohledu programátora se jedná o programovací prostředek) a je k dispozici ve formě knihoven v programovacích jazycích *C* a *Fortran*.
- Jedná se o poměrně nízkoúrovňový nástroj používající pro vzájemnou interakci procesů asynchronní zasílání zpráv přes vyrovnávací paměti
- Každý proces má jednu aktivní vyrovnávací paměť pro vysílání a jednu pro čtení. Vytváří se a ruší dynamicky.
- Klíčovou částí prostředí PVM je proces PVMD běžící na pozadí (démon) na každém počítači, který je zařazen do virtuálního multiprocesoru vytvořeného pro konkrétního uživatele.

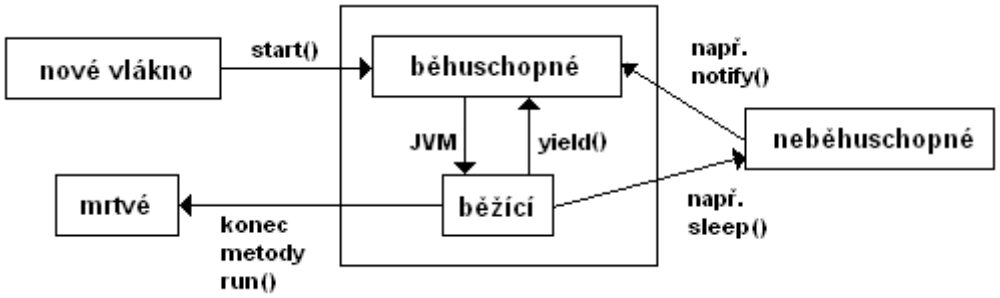
- Počítače použité v konfiguraci virtuálního multiprocesoru jsou v rámci PVM identifikovány svým síťovým jménem
- Na jednom stroji může být alokováno několik procesů jedné aplikace i několik démonů PVMD (pro jinou aplikaci).

4.2.3 MPI

- *MPI* je knihovna pro podporu paralelních výpočtů s systémech s distribuovanou pamětí.
- Je k dispozici pro jazyky *C* a *Fortran*.
- Není zaručen determinismus chování programu a překladač může provádět jen velmi omezené kontroly správného využití *MPI* funkcí.
- Oproti *PVM* se jedná o programovací prostředek vyšší úrovně, vztah mezi nimi je asi jako mezi jazykem symbolických adres (*PVM*) a vyšším programovacím jazykem (*MPI*). Lze říci, že v *PVM* lze naprogramovat „skoro cokoliv“, ale dá to velkou práci a program bude patrně nepřenositelný. Dominantní aplikací pro *MPI* jsou numerické výpočty s regulárními daty (vektory či matice s číselnými prvky), přičemž pro takové aplikace je programování relativně pohodlné a program je dobře přenositelný na jiné instalace *MPI*.

5 Programování v prostředí se sdílenou pamětí, vlákna v rámci aplikace (multithreading), požadavky na korektnost výpočtu, základní a strukturované formy interakce vláken

- V prostředí se sdílenou pamětí se nachází jeden a více procesorů, které jsou přes paralelní sběrnici připojeny na sdílenou paměť.
- Při programování v prostředí se sdílenou pamětí se často využívají vlákna, která na rozdíl od procesů sdílí adresní prostor. Jeden proces se může být rozdělen na více vláken
- Z hlediska systému se vícevláknová aplikace jeví jako jeden proces, chod aplikace je členěn do souběžně běžících vláken
- Stavy vlákna:



- Požadavky na korektnost výpočtu jsou následující. Správný program je:
  - Flow-correct – Program musí v každém běhu skončit a to vždy se stejným výsledkem
  - Logically-correct – Program navíc dává správný výsledek
- Jak docílit správnosti programu:
  - Co nejmenší a nejjednodušší interakce vláken
  - Řešení interakce nesmí činit předpoklady o vzájemné rychlosti vláken
  - Nepoužívat priority vláken, pokud to není nutné
  - Nezabíjet vlákna zvenku a obecně nezasahovat do jejich běhu

5.1 Primitivní formy interakce vláken

- **Synchronizace**
  - Zajištění návaznosti činností mezi vlákny
  - **Semafor**
    - Základní prostředek pro synchronizaci vláken
    - Obsahuje frontu, ve které čekají vlákna, čítač a dvě funkce, které volají vlákna před vstupem a po výstupu z kritické sekce – P() (Proberen – čekání na semaforu) a V() (Verhogen – signál na semaforu)
    - Při synchronizaci se z vlákna A volá P() a z vlákna B se volá V()
  - **Bariéra**
    - Prostředek pro synchronizaci více vláken v jeden okamžik
    - Typicky se používá, pokud více vláken provádí výpočet a pro další pokračování výpočtu je nutné dokončení dílčí části u všech vláken. Vlákna, která skončí dílčí výpočet čekají na bariéře, dokud výpočet neukončení poslední vlákno, pak mohou opět všechna pokračovat v činnosti

        - Zpravidla obsahuje čítač určující, kolik vláken se má ještě na bariéře zastavit. Dokud nedorazí všechna vlákna, všechny příchozí se uspí. Jakmile dorazí poslední vlákno, všechny vlákna čekající na bariéře se probudí
- **Sdílení dat**
    - Procesy využívají ke komunikaci datové struktury umístěné ve sdílené paměti.
    - Vlákna se navzájem nemusí znát, musí však být vyloučena současná změna dat více vlákny najednou (mutual exclusion).
    - Části programu, ve kterých může dojít ke konfliktním přístupům k datům se označují jako *kritické sekce*.
    - **Binární semafor**
        - Stačí pro vzájemné vyloučení přístupu k datům.
        - Zaručí, že k datům může najednou přistupovat pouze jedno vlákno
    - **Zámek**
        - Funguje podobně jako binární semafor, ale místo fronty uspaných vláken používá aktivní čekací smyčku
    - Použití zámků či semaforů může vést k zablokování aplikace.
    - Zablokování lze zabránit očíslováním zdrojů a zamykáním v určeném pořadí
- **Zasílání zpráv**
    - Na rozdíl od předchozích forem komunikace je použitelná i pro systémy s distribuovanou pamětí
    - Je třeba realizovat vyrovnávací paměť pro zprávy (fronta zpráv, send, receive)
    - Podle charakteru send a receive lze komunikaci rozdělit na:
        - Asynchronní – blokující je pouze receive
        - Synchronní – blokující je send i receive
    - Podle adresování zpráv lze komunikaci rozdělit na:
        - Symetrické – Zpráva obsahuje jak adresu příjemce tak i odesílatele
        - Asymetrické – Zpráva obsahuje jek adresu příjemce
        - Nepřímé – Zpráva obsahuje pouze adresu vyrovnávací paměti či komunikačního kanálu, tj. odesílatel a příjemce se navzájem vůbec nemusí znát

5.2 Strukturované formy interakce vláken

- Oproti primitivním formám komunikace poskytují větší bezpečnost programování a navíc bývají často přímo implementovány v programovacím jazyce
- **Monitor**
    - Objekt poskytující vláknům služby pro konkurenční prostředí
    - Typový monitor – objektový typ (class), podle kterého se vytváří instance
    - Model výpočtu pomocí monitoru
        - Vlákna se mezi sebou vůbec neznají, každé zná monitor
        - Veškeré interakce vláken probíhají nepřímo prostřednictvím monitoru
    - Implementace monitoru obsahuje frontu, zámek, sdílená data procesů a služby monitoru (metody)
    - Vlastnosti:
        - Volání metody (služby) musí být atomické (nejde přerušit z venku, musí být zajištěna konzistence dat)
        - Implementace atomicity vyžaduje zámek
        - Někdy nelze službu poskytnout → uspat vlákno (blokující volání služby), nebo vrátit error code

        - Pro blokující volání mývají monitory podporu v podobě funkcí *wait()* (blokující) a *notify(), notifyAll()* (neblokující)
        - Je potřeba fronta čekajících vláken
        - Ve *wait()* je nutno odemknout zámek, za *wait()* zase zamknout (v monitoru smí být aktivní pouze jedno vlákno)
        - V Javě je jen 1 fronta vláken, ale ta mohou čekat z různých důvodů – volat *notifyAll()* a vlákna opět otestují možnost běhu
        - V POSIX vláknech je fronta implementována podmínkovou proměnnou
- **Rendezvous**
    - Používaná forma je *synchronní* (vlákna se při rendezvous synchronizují) a *asymetrická* (1 vlákno v roli klient, druhé v roli server – akceptuje rendezvous, klient musí znát (mít odkaz na) server, naopak ne).
    - V každém vlákně se nachází jeho výkonný kód, v serveru se navíc nachází společný kód vláken, který se provede po zavolání *entry* (volá klient) a jeho akceptaci (provádí server)

6 Programové prostředky pro multithreading: Ada – tasky a rendezvous, Java – vlákna a monitory, rozhraní POSIX pro vlákna v jazyce C – charakteristika

6.1 Ada – tasky a rendezvous

- Paralelně proveditelná aktivita (proces) se v Adě nazývá *task*.
- Pro synchronizaci se používá rendezvous

6.1.1 Tasky

- Konstrukce task představuje program paralelně proveditelného procesu, schopného komunikace s ostatními procesy
- Deklarace je následující:

```
task jméno is
     deklarace jmen komunikačních typů
end jméno;
task body jméno is
     lokální deklarace a příkazy
end jméno;
```

- Pro ukončení procesu lze použít příkaz `abort jméno;`, ale jeho použití by mělo být výjimečné

6.1.2 Rendezvous

- Používaná forma je *synchronní* (vlákna se při rendezvous synchronizují) a *asymetrická* (1 vlákno v roli klient, druhé v roli server – akceptuje rendezvous, klient musí znát (mít odkaz na) server, naopak ne).
- Schéma rendezvous dvou vláken:

```
a (klient)   b (server)
    |            |
    |            |
--------------------------- synchronizace
* b.P(Param)    accept P(Param)  \
    |           begin            |
    |           …                |- entry
    |           …                |
    |           end;             /
    |            |
```

* - volání entry
- funkce P má přístup k datům obou vláken
- Jak se pomocí rendezvous zařídí „pořádný server“ (monitor) – v Adě 83 je navíc příkaz *select*, tím lze vytvořit skupinu volání entry.

```
select
  accept P1() begin … end
or
  accept P2() begin … end
end select
```

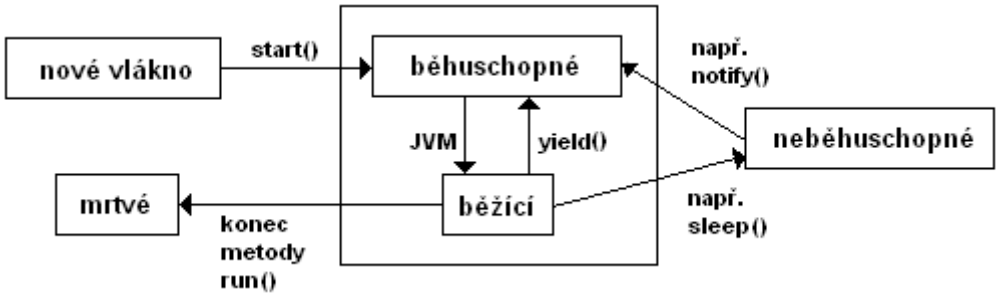
- Tento select by měl být ve smyčce (loop) a tím máme dobrý server, který může obsluhovat více metod.

6.2 Java – vlákna a monitory

- Prostředky pro práci s vlákny jsou součástí standardních knihoven jazyka Java. Klíčovou je třída `Thread`, jejíž instance (a instance potomků) představují vlákna programu.
- Pro synchronizaci se používá monitor, který je součástí každého objektu v Javě

6.2.1 Třída Thread

- Tvoří základ všech paralelních programů v Javě.
- Pro jednoduché programy stačí oddědit od této třídy a překrýt metodu *run()*, do které se napíše výkonný kód vlákna.
- Protože někdy je potřeba, aby naše třída dědila od jiné a zároveň měla vlastnosti vlákna, existuje ještě rozhraní *Runnable*, které stačí implementovat a třída rovněž získá vlastnosti vlákna. Tento postup je však méně častý.
- Pro napsání paralelního programu stačí vytvořit potřebné třídy, napsat jejich výkonné kódy do metod *run()* a pak vlákna spustit (vlákna se spouští metodou `start()`, nerozběhnou se tedy hned po vytvoření)
- Vlákno se může potenciálně nacházet v jednom z pěti stavů, které si zde nyní podrobně popíšeme:
    - **Nové vlákno –** Vlákno bylo vytvořeno, ale dosud nebylo spuštěno metodou *start()*.
    - **Běhuschopné –** Metoda *start()* už proběhla; těchto vláken může být více, ale na jednoprocesorovém stroji je vždy jen jedno **běžící**, ostatní musí čekat na předání řízení.
    - **Neběhuschopné –** Vlákno, které bylo uspáno metodou *sleep()*, nebo čeká na *wait()* (viz dále), nebo čeká na I/O.
    - **Mrtvé vlákno –** Vlákno, jehož metoda *run()* již skončila.



6.2.2 Monitory v Javě

- Komunikace vláken je řešena přes sdílenou paměť, v programu se tedy vyskytují kritické sekce.
- Pro veškerou komunikaci a synchronizaci vláken v Javě se využívá monitor, který je součástí každého objektu
- Pro implementaci monitorů jsou uvnitř objektu monitoru (= má synchronizační metody) skryté atributy a to:
    - 1 zámek – pro všechny synchronizované metody
    - 1 fronta – pro zabrzdění vláken, kterým nemohla být poskytnuta služba
    - Nad frontou fungují privátní metody monitoru *wait(), notify(), notifyAll()*.

6.3 C – POSIX

- Jde o knihovnu pro jazyk *C* umožňující práci s vlákny.

- Obsahuje tři základní typy objektů – *vlákna*, *mutexy (zámky)*, *podmínkové proměnné*, které jsou reprezentovány pomocí tzv. handle (zobecněný ukazatel) *pthread_t, pthread_mutrex_t*, *pthread_cond_t*.
- Kromě handlů existují ještě tzv. atributované objekty (k popisu vlastností vláken, mutexů, podmínkových proměnných) *pthread_attr_t, pthread_mutexattr_t*, *pthread_condattr_t*.
- Vytváření a rušení objektů je dynamické, každé vlákno má svůj zásobník, tj. proměnné definované v programu vlákna (paměťová třída auto → zásobník) jsou lokální.
- Proměnné definované v hlavním programu jsou globální (sdílené a musejí se zamykat)

6.3.1 Vlákna

- Program vlákna je vlastně funkce *C* s typem `void* prog_name(void* arg).`
- Vlákno se vytvoří následujícím způsobem:
```
int status;  //0 uspech, jinak chyba
status = pthread_create(&worker, NULL, prog_name,…);
```
- NULL znamená ukazatel na atributy objektu, pokud je to NULL, vytvoří se automaticky atributový objekt s implicitním nastavením.
- Vlákno běží hned po vytvoření.
- Stavy vlákna jsou *ready*, *running*, *waiting* a *terminated*. Vlákno se ruší pomocí *detach*.
- *Atributy:*
    - způsob plánování
        - FIFO – nejprioritnější kategorie (nejdříve se plánují vlákna této kategorie)
        - RR – round-robin
        - FG – foreground, implicitní kategorie, střídání vláken, ty s vyšší prioritou mají více času
        - BG – background, všechna vlákna se střídají, ale dostávají méně času než FG
    - priorita
    - rozměr zásobníku
    - hlídač zásobníku – jak daleko se můžeme od konce zásobníku dostat, jinak vznikne výjimka
    - konec vlákna
        - vlákno dojde na konec svého programu. Na toto ukončení se lze synchronizovat z jiného vlákna pomocí *pthread_join(na_koho_se_čeká, kam_přijde_výsledek)* .
        - zabito z vnějšku – pokud možno nepoužívat. Základní funkce pro likvidaci *pthread_cancel(oběť)*; oběť se může bránit *pthread_setcancelstate(...)*. K likvidaci nemůže dojít kdekoliv v kódu vlákna, ale jen v předem připravených místech (volání blokující funkce, volání *pthread_testcancel()*). Popisovaný způsob je synchronní, existuje i asynchronní. Existuje i havarijní destruktor volaný při rušení

6.3.2 Mutexy (zámky)

- Exitují k ochraně globálních dat, ne pro synchronizaci. K synchronizaci lze vyrobit semafory.
- Vytvoření mutexu je jednoduché:
```
pthread_mutex_t zamek = PTHREAD_MUTEX_INITIALIZE;
```
- Existují tři typy zámků

    - normální – konkrétní vlákno ho může zamknout jen 1x
    - rekurzivní – konkrétní vlákno ho může zamknout vícekrát (pro rekurzivní zpracování globálních dat)
    - ladící – ERRORCHECK; Pozná se opakované zamčení
- Základní funkce:
    - `pthread_mutex_lock(&zamek);`
    - `pthread_mutex_unlock(&zamek);`
    - `pthread_mutex_try_lock()` – neblokující zamykání

6.3.3 Podmínkové proměnné

- Implementují frontu, kde vlákna čekají na splnění podmínky, neimplementují test podmínky (ten musím být v kódu vlákna).
- Aby test podmínky + případná změna proměnné byla atomická akce, je třeba sdružit podmínkovou proměnnou s mutexem.
- Typy podmínkových proměnných:
    - `pthread_cond_wait(&podm_prom, &mutex);` - čekající vlákno musí odemknout mutex
    - `pthread_cond_signal(&podm_pom);` - jako *notify()*
    - `pthread_cond_broadcast(&podm_prom);` - jako *notifyAll()*

7 Programování v paralelním výpočetním prostředí s distribuovanou pamětí – charakteristika a parametry prostředí, principy realizace základních modelů paralelního výpočtu

- V zásadě existují dva typy multiprocesorů s distribuovanou pamětí:
    - Počítačové sítě (též vícepočítačové či multipočítačové systémy)
        - Jsou vytvořeny spojením několika (většinou různých) počítačů komunikačními linkami
        - Jsou místně rozlehlé
        - Při použití vhodného programového prostředku pro spojení počítačů se pak každá stanice může tvářit jako jeden procesor multiprocesorového stroje
        - Pro tento přístup se nečastěji využívá PVM
    - Paralelní počítače
        - Jsou vytvořeny spojením většího počtu výpočetních a dalších prvků do jednoho funkčního celku
        - Jsou místně kompaktní (např. v jedné skříni) a zpravidla homogenní (výpočetní prvky stejného typu)
        - Pro psaní paralelních programů se často využívá MPI
- Protože neexistuje sdílená paměť, používá se pro komunikaci mezi procesy především zasílání zpráv
- Protože systémy s distribuovanou pamětí nemají žádný úzký profil ve formě sběrnice, přes kterou by procesory přistupovali ke sdílené paměti, hodí se pro úlohy vyžadující tzv. masivní paralelismu (stovky až tisíce procesorů). Nejčastěji používaným modelem je proto SPMD.
- U této otázky by možná bylo vhodné také charakterizovat modely SPMD, MPMD a MPSD, viz samostatná otázka 2.

7.1 SPMD

- Nejčastěji používaný model v prostředí s distribuovanou pamětí
- Do procesorů se zavede množina procesů pracujících podle jednoho programu a každá dostane ke zpracování část objemných dat (části dat jsou strukturně stejné, ale hodnotami různé)
- Procesy si vyměňují informace (většinou zasílání dílčích výsledků) zasíláním zpráv
- Většinou se používá přístup *farmer-workers*, tedy jeden řídící proces a *n* dělníků
    - Řídící proces rozdělí úlohu na části, které předá dělníkům a následně pak agreguje dílčí a vytváří globální výsledky
    - Dělníci dostanou přidělen úsek práce a vytváří dílčí výsledky, které předávají řídícímu procesu
    - Program hlavního procesu a dělníka může být oddělen do dvou samostatných souborů (typicky PVM), nebo jeden zdrojový soubor obsahuje jak program dělníka, tak hlavního procesu a určení, co má který proces dělat, se provádí až během výpočtu (typicky MPI)
- Problém rozdělování práce
    - Rozdělování práce mezi procesy je možné buď staticky podle počtu použitých procesorů, nebo dynamicky.

- Při statickém rozdělení se vytvoří tolik procesů, kolik je k dispozici procesorů a každý proces dostane stejný kus práce. Problémem je, obzvláště při použití PVM, neznámé zatížení jednotlivých procesorů a výsledek je tedy znám až po skončení procesu na nejpomalejším procesoru
- Při dynamickém rozdělování se vytvoří více procesů než je procesorů a dynamicky se přidělují na procesory v závislosti na vytížení jednotlivých procesorů. Výhodou je nezávislost na nejpomalejším procesoru, nevýhodou je zřejmá komunikační režie výpočtu.

7.2 MPMD

- V tomto případě existují různé procesy pracující podle různých programů nad strukturně jinými daty
- Procesy mezi sebou komunikují zasíláním zpráv
- Do tohoto modelu lze zařadit i řetězové zpracování MPSD, při kterém si procesy probíhající podle různých programů „předávají“ ke zpracování datové záznamy

8 Charakteristika a porovnání výpočetních nástrojů PVM a MPI, příklady použití

- Oba nástroje, PVM (Parallel Virtual Machine) i MPI (Message Passing Interface) se používají v prostředí s distribuovanou pamětí

8.1 PVM

- *PVM* je v nejobecnějším pohledu *univerzální výpočetní model* pro paralelní programování, který má dobrou naději stát se mezinárodním standardem.
- Je implementováno pro paralelní počítače různého typu (tj. z pohledu programátora se jedná o programovací prostředek) a je k dispozici ve formě knihoven v programovacích jazycích *C* a *Fortran*.
- Uplatňuje se především v počítačových sítích, kdy z různorodých propojených počítačů se vytvoří virtuální multiprocesorový počítač
- Jedná se o poměrně nízkoúrovňový nástroj používající pro vzájemnou interakci procesů asynchronní zasílání zpráv přes vyrovnávací paměti. Každý proces má jednu aktivní vyrovnávací paměť pro vysílání a jednu pro čtení. Vytváří se a ruší dynamicky.
- Klíčovou částí prostředí PVM je proces PVMD běžící na pozadí (démon) na každém počítači, který je zařazen do virtuálního multiprocesoru vytvořeného pro konkrétního uživatele.
- Počítače použité v konfiguraci virtuálního multiprocesoru jsou v rámci PVM identifikovány svým síťovým jménem. Na jednom stroji může být alokováno několik procesů jedné aplikace i několik démonů PVMD (pro jinou aplikaci).
- K manuálnímu ovládání virtuálního multiprocesoru je k dispozici PVM konzola. Její příkazy jsou následující:
    - *add host_name* – Přidá stroj se síťovým jménem host_name do konfigurace virtuálního multiprocesoru
    - *spawn* – spustí výpočet připravené aplikace
    - *conf* – Vypíše konfiguraci virtuálního multiprocesoru (jména typ strojů, identifikátory procesů PVMD,…)
    - *mstat host_name* – Vypíše stav specifikovaného stroje
    - *pstat proces_id* – vypíše stav specifikovaného procesu
    - *ps –a* - vypíše všechny procesy běžící aplikace, jejich alokaci na stroj a identifikátory
    - *sig signal_num* – Pošle signál procesům, lze je pomocí toho příkazu takto ovládat
    - *kill process_id* – Ukončení libovolného procesu aplikace
    - *quit* – Ukončí výpočet
- Pro paralelizaci výpočtu existuje množství metod, všechny mají předponu `pvm_`.
- Paralelní program je typicky rozdělen na hlavní proces, který rozděluje práci a řídí celý výpočet a dělníky, kteří provádějí dílčí výpočty.
- Řízení dělníků a předávání dat se děje pomocí zasílání zpráv

8.2 MPI

- *MPI* je knihovna pro podporu paralelních výpočtů s systémech s distribuovanou pamětí.
- Je k dispozici pro jazyky *C* a *Fortran*.

- Není zaručen determinismus chování programu a překladač může provádět jen velmi omezené kontroly správného využití *MPI* funkcí.
- Oproti *PVM* se jedná o programovací prostředek vyšší úrovně, vztah mezi nimi je asi jako mezi jazykem symbolických adres (*PVM*) a vyšším programovacím jazykem (*MPI*). Lze říci, že v *PVM* lze naprogramovat „skoro cokoliv“, ale dá to velkou práci a program bude patrně nepřenositelný. Dominantní aplikací pro *MPI* jsou numerické výpočty s regulárními daty (vektory či matice s číselnými prvky), přičemž pro takové aplikace je programování relativně pohodlné a program je dobře přenositelný na jiné instalace *MPI*
- Další vlastnosti MPI můžeme shrnout do několika bodů:
    - *MPI* je primárně určeno pro homogenní výpočetní prostředí, takže poskytuje prostředky i pro „synchronní“ algoritmy (tj. takové, kde se předpokládá přibližně stejná rychlost běhu jednotlivých procesů) a odpovídající statické rozdělení práce mezi procesy.
    - *MPI* primárně využívá *SPMD* model paralelního výpočtu, tj. vyrobí se, na rozdíl od *PVM*, jen jeden spustitelný soubor programu a ten se zavede do zvoleného počtu procesorů.
    - Komunikace procesů je asynchronní zasílání zpráv s přímým adresováním přes číselné ID procesu.
    - *MPI* má svoje „primitivní datové typy“ a z nich lze skládat „strukturované typy“.
    - Existuje sada funkcí pro tzv. „globální operace“, tj. operace nad daty, jejichž instance jsou rozprostřeny ve všech procesech výpočtu.
- Přestože se vytváří jen jeden spustitelný soubor, program je většinou, podobně jako v PVM, členěn na hlavní proces rozdělující práci a agregující dílčí výsledky a pracovní procesy, které počítají dílčí výsledky.
- Členění je provedeno přímo v programu, který tedy musí obsahovat jak kód hlavního procesu, tak dělníků rozlišení se provádí podle čísla procesu (každý proces má unikátní číslo, procesy jsou číslovány od nuly, typicky 0 je hlavní proces).
- Protože MPI je primárně určeno pro zpracování rozměrných polí a matic dat, obsahuje funkce pro rozdělení dat mezi pracovní procesy i funkce pro snadné získání globálních výsledků
- Na rozdíl od PVM se používá spíše na fyzických multiprocesorech, přičemž počet procesorů použitých ve výpočtu se zadává při spuštění programu