Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Plánovač vláken SAN

Plzeň, 2011 Jiří Nohavec

Originál zadání

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 19. 5. 2011

……

Jiří Nohavec

Abstract

In a response to shortcomings of IP-based networks, DARPA proposed the concept of Active Networks. Active Networks presents a new concept of applications and services in computer networks. Packet was associated with a program code that executes, as the packet is being delivered through the network. This allows rapid design, deployment and co-existence of network services and protocols.

Active Networks brings the need to manage code execution in a specific manner, to conserve server resources. In diploma thesis, I focus on scheduling. In the standard fashion of operating-system scheduling, like Windows or Linux scheduler, possibly too many threads could be in the runnable state. This would lead to exhaustion of system resource and performance degradation. A different approach is needed.

The scope of this thesis covers scheduler, lightweight fiber execution, synchronization, deadlock detection and prevention. For Smart Active Node, I implemented a proof of the concept to demonstrate the chosen approach to scheduling. Smart Active Node is active-networking project at Department of Computer Science and Engineering on University of Western Bohemia.

Keywords: SAN, Smart Active Node, active network, scheduler, scheduling, fiber thread, synchronization, deadlock detection, deadlock prevention.

Poděkování

Chtěl bych tímto poděkovat všem vyučujícím Katedry informatiky a výpočetní techniky, kteří se nám, studentům, poctivě a pečlivě po dobu našeho studia věnovali, předávali nám své znalosti a zkušenosti z oboru informačních technologií a umožnili nám účastnit se a spolupracovat na různých a zajímavých projektech.

Obzvláště bych pak chtěl poděkovat Tomáši Koutnému, Ph.D. za vedení této diplomové práce a umožnění se realizovat a přispět svým dílem do problematiky aktivních sítí.

Obsah

1 Úvod

Počítačové sítě se stali nedílnou součástí lidské společnosti. Umožňují nám komunikovat s ostatními uživateli, vyhledávat a získávat informace, efektivně pracovat a také se bavit.

Aby však počítačová síť, nebo obecně jakákoliv komunikační síť, mohla fungovat a poskytovat uživatelům své služby, je potřeba nejprve zajistit, aby všechna zařízení byla navzájem propojena a uměla spolu komunikovat.

Stejně jako lidé různých národností mluví různými jazyky a navzájem se mohou jen těžko bez tlumočníka domluvit, tak i zařízení připojená k počítačové síti musí být schopná si navzájem porozumět. Ze signálů přijatých z komunikačního kanálu musí umět sestavit odpovídající informaci, kterou zaslala druhá strana.

Toto je zajištěno pomocí takzvaných komunikačních protokolů. Komunikační protokol je soubor pravidel, jak zacházet se signály a jak ze signálů získat odpovídající informaci. Aby spolu mohla zařízení komunikovat pomocí určitého protokolu, je potřeba tato zařízení vybavit odpovídajícím programovým vybavením, které pravidla protokolu implementuje.

Zařízení připojená do počítačové sítě můžeme rozdělit do dvou hlavních skupin:

- Koncové uzly – Servery, pracovní stanice (PC). Tyto stroje jsou vybavené plnohodnotnými operačními systémy. Uživatelé mají umožněnu instalaci a provoz libovolného programového vybavení.
- Mezilehlé uzly – Směrovače (routery), přepínače (switche). Tato jednoúčelová vestavěná (embedded) nebo SoC (System on Chip) zařízení mají své programové vybavení zabudováno přímo výrobcem. Uživatelům není instalace nových programů a protokolů obvykle umožněna. Výrobce těchto zařízení je nucen se řídit síťovými standardy, jejichž prosazení, schválení a nasazení trvá řádově roky. Některé složitější směrovače mohou obsahovat vlastní operační systém.

Protože programové vybavení mezilehlých uzlů nelze jednoduše upravovat, počítačová síť poskytuje pouze předem připravenou omezenou množinu služeb. Síť se chová staticky, umožňuje jen zasílat data mezi jednotlivými službami, avšak již neumožňuje tyto služby zavádět a spravovat. Uživatelské aplikace si nemohou vytvářet vlastní služby přímo na míru, ale musí se spokojit s omezenou množinou standardních služeb.

1.1 Tradiční sítě

V tradiční IP síti přenášejí zasílané zprávy (pakety) jen data. Předpokládá se, že služby, které data zpracovávají, jsou již na jednotlivých uzlech zavedeny a spuštěny. V případě, že služba není na uzlu k dispozici, je doručený paket zahozen, protože uzel nemá

možnost data doručená v neznámém formátu interpretovat. Odesílatel se o nedoručení své zprávy dozví se zpožděním až po vypršení časového limitu na doručení odpovědi.

Jednotlivé služby jsou identifikovány pomocí čísla procesu implementujícího daný protokol nebo pomocí portu protokolů TCP a UDP. Avšak není nijak zajištěna pevná vazba konkrétní služby na daný port nebo číslo procesu. Stejně tak není nijak zajištěna konkrétní verze služby. Pokud na daném portu běží jiná než očekávaná služba, budou data doručena této nesprávné službě.

Tyto problémy je možné alespoň částečně vyřešit programově v rámci dané služby, ale díky tomu zbytečně roste složitost řešeného problému. Obvykle na úkor samotné poskytované služby.

1.2 Aktivní sítě

Jako reakce na nedostatky tradičních počítačových sítí byl v roce 1995 započat organizací DARPA výzkum nového konceptu počítačových sítí – aktivní sítě.

Výzkum aktivních sítí si kladl za cíl odstranit výše zmíněné neduhy tradičních sítí. Cílem celého projektu bylo vytvoření dynamické sítě, jejíž chování si budou moci uživatelé zcela přizpůsobit svým potřebám, tedy snadno vytvářet a nasazovat nové služby napříč celou sítí bez nutnosti ručně zasahovat do programového vybavení jednotlivých uzlů sítě.

Základní myšlenkou aktivních sítí je možnost spolu s daty ve zprávách zasílat také samotné služby. Tyto kombinované zprávy se v terminologii aktivních sítí nazývají kapsule. Kapsule tedy přenáší spolu s daty také novou funkcionalitu sítě.

Služba ve světě aktivních sítí představuje kód programu, který je dle potřeby spouštěn na jednotlivých uzlech sítě. Každá služba je v rámci aktivní sítě jednoznačně identifikována, včetně jednotlivých verzí. Každá kapsule odpovídá právě jedné konkrétní službě a nehrozí konflikty verzí nebo nedostupnost služby.

Jednotlivé síťové uzly budou schopny se dynamicky naučit nové služby kdykoliv, kdy to bude uživatel vyžadovat. Nové služby budou moci být snadno vyvíjeny a okamžitě nasazeny a otestovány [TW02].

Další důležitou vlastností aktivních sítí je důraz na bezpečnost. Protože uzel aktivní sítě bude vykonávat cizí programový kód, je nutné tento kód během vykonávání kontrolovat, aby nenarušoval činnost ostatních aplikací, nespouštěl nevhodné služby nebo nespotřebovával příliš mnoho systémových prostředků. Bezpečnost může být dále řešena pomocí uživatelských účtů, certifikátů zajišťujících věrohodnost autora aplikace, anebo interpretací kódu a kontrolou jednotlivých příkazů a volání metod.

1.3 Rozdíl mezi tradičním a aktivním uzlem sítě

Uzly tradiční sítě se obvykle starají jen o sestavování směrovací tabulky, přečtení cílové adresy zprávy a výběru nejvhodnější cesty pro její doručení. Funkcionalita těchto uzlů je velmi omezená, obvykle postačuje pevná implementace několika směrovacích protokolů a protokolů pro rezervaci a řízení služeb (QoS – Quality of Service). Tato zařízení mohou mít vlastní vestavěný operační systém, ale jeho možnosti jsou velmi omezené.

Uzly aktivní sítě by měly být schopny navíc vykonávat programový kód jednotlivých kapsulí a služeb, a tím tak měnit a rozšiřovat svoji funkcionalitu. Dále by uzly měly umožnit vykonávanému kódu služby přístup ke směrování a řízení svých služeb. Tímto je možné například umožnit kapsuli, aby si sama určovala svoji cestu sítí.

1.4 Vykonávání uživatelských aplikací v aktivních sítích

Každý aktivní uzel musí být vybaven prostředky pro vykonávání uživatelských aplikací, takzvaných běhových prostředí (EE – Execution Environment). Tato prostředí mají zajistit izolované a zabezpečené vykonávání aplikací a zároveň umožnit těmto aplikacím přístup k ovládacímu rozhraní serveru. Uzel aktivní sítě může být vybaven různými běhovými prostředími pro aplikace a kapsule napsané v různých jazycích.

1.5 Plánování uživatelských aplikací v aktivních sítích

K vykonávání kódu je potřeba procesor a paměť. Pro ukládání kódu je třeba úložiště. Procesory, paměť a úložiště můžeme považovat za systémové zdroje uzlu, které je nutné řídit, spravovat a přidělovat uživatelským programům. K tomu je potřeba mezivrstva řídícího (operačního) systému umístěná mezi zdroji (hardwarem uzlu) a uživatelskými aplikacemi.

Dalším důvodem pro správu zdrojů je jejich omezenost. Uzel bude mít vždy omezený počet procesorů a omezenou velikost paměti. Řídicí systém proto musí přidělovat a odebírat systémové zdroje, aby bylo v nejlepší možné míře umožněno vykonávání všech uživatelských aplikací. Například v jeden okamžik obvykle bude spuštěno více aplikací, než je fyzický počet procesorů. Řídicí systém musí zajistit všem spuštěným aplikacím přístup k procesoru, ale nemůže tak učinit okamžitě pro všechny. Jednotlivé aplikace se budou o dostupné procesory postupně střídat.

Řídicí systém aktivního uzlu musí poskytovat následující funkce:

- Spravovat systémové prostředky a přidělovat je aplikacím.
- Poskytovat aplikacím abstraktní rozhraní pro přístup k hardwaru uzlu.
- Umožnit aplikacím přístup ke službám uzlu – spouštět nové aplikace, odesílat a přijímat kapsle, umožnit kapslím přístup ke směrování.

Požadavky na řídicí systém uzlu aktivní sítě jsou velmi podobné požadavkům na běžný operační systém z osobních počítačů.

Součást řídicího systému uzlu aktivní sítě, která rozhoduje o přidělování a odebírání procesorového času jednotlivým aplikacím a jejich spouštění a ukončování, se nazývá plánovač. Plánovač pracuje na základě dané plánovací strategie, která se snaží optimalizovat přístup k systémovým zdrojům pro jednotlivé běžící aplikace a zajistit tak jejich přerozdělení.

Tato diplomová práce je zaměřena právě na plánovač a plánování uživatelských aplikací pro uzly aktivní sítě.

2 Cíle a motivace

Uzel aktivní sítě SAN (Smart Active Node) používá k vykonávání uživatelských aplikací vlastní interpretr Java bytecode. K zajištění paralelního běhu aplikací plánovač uzlu SAN využívá standardní vlákna a synchronizační prostředky platformy Java, každá aplikace je spuštěna v novém vlákně a o její plánování a synchronizaci se stará virtuální stroj Javy (JRE – Java Runtime Environment).

Úkolem této diplomové práce je upravit plánovač a interpretr uzlu SAN tak, aby veškeré plánování a synchronizaci aplikací prováděl plánovač uzlu SAN sám a bez nutnosti spoléhat se na standardní prostředky JRE. Za tímto účelem bude nutno implementovat vlastní fiber vlákna (vlákna v uživatelském prostoru nevyžadující podporu jádra operačního systému) a systém pro jejich správu a plánování.

2.1.1 Očekávané výhody použití fiber vláken

Důvody k přechodu od standardních JRE vláken k implementaci, správě a plánování vlastních fiber vláken jsou následující:

- Vlastní řízení priority vykonávání uživatelského programového kódu, zavedení prioritních tříd. Například kapsule aplikace pro měření odezvy sítě (ping) by měla být vždy vykonána přednostně.
- Snížení paměťových nároků. Pro každou aplikaci nebude potřeba vytvářet novu instanci interpretru.
- Snížení nároků na systémové prostředky. Pro každou aplikaci nebude potřeba vytvářet nové vlákno.
- Urychlení zavádění uživatelských aplikací, jednotlivé aplikace nebudou spouštěny v nových vláknech, ale budou se jen střídat o jedno neustále běžící vlákno interpretru, které bude spuštěno při startu systému.
- Možnost spuštění více interpretrů paralelně za účelem využití potenciálu vícejádrových procesorů.
- Možnost vytvoření vlastní implementace synchronizačních prostředků za využití vlastního plánování fiber vláken. Synchronizační prostředky mohou být navíc vytvořeny na míru potřebám aplikací aktivních sítí.

2.1.2 Postup pro dosažení cílů této práce

Pro dosažení výše zmíněného cíle, tedy implementaci fiber vláken pro projekt SAN, bude použita posloupnost následujících kroků:

- Prozkoumání existujících řešení aktivních sítí a jejich přístupu k plánování a vykonávání uživatelského programového kódu.
- Navržení architektury fiber vláken.
- Navržení vlastního plánování fiber vláken, včetně možnosti využití více paralelně běžících instancí interpretru.
- Navržení synchronizačních prostředků pro fiber vlákna.

- Implementace navržených řešení pro projekt SAN.
- Rozšíření služeb uživatelského rozhraní uzlu SAN o nově vzniklou funkcionalitu a její vhodná dokumentace.
- Vytvoření ukázkových aplikací pro účely ověření a demonstrace nové funkcionality uzlu SAN.

3 Související práce

Cílem první části této kapitoly je prozkoumat existující řešení aktivních sítí a jejich přístupu k plánování a vykonávání programového kódu uživatelských aplikací.

3.1 ANTS

ANTS (Active Node Transport System) je implementace aktivní sítě založená na programovacím jazyce Java. Jejím cílem bylo ověřit smysluplnost konceptu aktivních sítí [WGT98]. Projekt byl započat v roce 1998 a poslední aktualizace byla provedena v roce 2002.

Uživatelské aplikace nebo kapsule jsou zde reprezentovány třídou v jazyce Java. Tato třída je odvozena od dané abstraktní třídy, která slouží jako rozhraní pro činnost aplikací a kapsulí. Každá aplikace nebo kapsule je spuštěna v novém vlákně, které spravuje virtuální stroj, ve kterém ANTS běží. Není použito standardní JRE, ale vlastní implementace virtuálního stroje. Tato vlastní implementace se jmenuje Janos Java Node OS.

Ačkoliv Janos poskytuje vlastní rozhraní pro používání vláken, tato vlákna sám neimplementuje. Rozhraní vláken jen zapouzdřuje prostředky standardního JRE, které se na pozadí stará o správu a plánování vláken. Vlastní řešení vláken a plánovače není v Janos implementováno, ačkoliv byla připravena potřebná rozhraní. Poslední aktualizace Janos byla provedena v roce 2001.

3.2 PAN

PAN (Practical Active Network) je experimentální implementace aktivních sítí. Cílem tohoto projektu byl výzkum praktického nasazení aktivních sítí a jejich potencionálního výkonu [Nyg98].

Uzel PAN je implementován v jazyce C/C++. Uživatelské aplikace a kapsule jsou napsány v libovolném jazyce, který je možné zkompilovat do x86 kódu. Tento kód je následně distribuován mezi jednotlivými uzly sítě a dle potřeby spouštěn přímým zavedením do paměti a nastavením odpovídajícího instrukčního ukazatele (program counter).

PAN nepodporuje vlákna a souběžný běh více uživatelských aplikací nebo kapsulí. Plánování je zajištěno pomocí algoritmu podle pořadí přístupu (FCFS).

3.3 Bowman Node OS

Bowman Node OS je rozšiřitelná platforma pro aktivní sítě určená pro práci s virtuálními proudy a kanály podobnými ATM [MBZC00].

Uživatelské aplikace mohou být napsány v různých programovacích jazycích, pro každý z nich je v uzlu připraveno odpovídající běhové prostředí.

Z hlediska plánování a vykonávání aplikací se Bowman spoléhá na operační systém. Plánování a spouštění uživatelských aplikací je zcela v režii operačního systému. Pro přístup k vláknům a jejich synchronizaci je použito rozhraní POSIX.

3.4 ANN

ANN (Active Network Node) je projekt určený pro implementaci distribuovaného výpočetního prostředí pomocí konceptu aktivní sítě [DPP98].

Uživatelské aplikace a kapsule se v terminologii ANN jmenují pluginy. Plugin je objektový x86 kód metody aplikace, který je šířen mezi uzly. Po přijetí kódu je metoda dynamicky zavedena do paměťového prostoru uzlu. Při přijetí požadavku na spuštění je metoda spuštěna komponentou zavaděče metod (Function Dispatcher). Zavaděč metod periodicky spouští metody patřící běžícím aplikacím nebo kapsulím. Každá metoda má přidělenou prioritu, která určuje periodu, s jakou má být zavaděčem metod spouštěna.

Chování metod sleduje komponenta správce zdrojů (Resource Controller). Pokud některá metoda spotřebovává příliš mnoho procesorového času, snižuje správce zdrojů její prioritu. Pokud naopak vlákno procesor příliš nevyužívá, její priorita je zvětšována. Komponenta správce zdrojů tedy využívá algoritmu prioritního plánování.

3.5 Smart Packets

Smart Packets je prototyp určený pro demonstraci možností aktivních sítí. Celý projekt je napsán v jazyce Java [KMHW98].

Uživatelské aplikace a kapsule jsou napsány v jazyce Java jako třída splňující určité rozhraní. Aplikace a kapsule jsou následně v aktivní síti distribuovány jako bytecode. Při spuštění je tento bytecode nahrán pomocí standardního class loaderu a spuštěn v novém vlákně JRE. Vlákna jsou plánována standardním plánovačem virtuálního stroje Java.

Smart Packets obsahuje komponentu správce zdrojů. Tato komponenta sleduje chování vláken a upravuje jejich prioritu pomocí plánovacího algoritmu víceúrovňové fronty se zpětnou vazbou (MLFQ – multi-level feedback queue). Tímto je komponenta správce zdrojů schopna v omezené míře ovlivňovat běh aplikací a můžeme ji považovat za plánovač.

3.6 Získané poznatky

Zkoumané implementace aktivních sítí využívají různé metody plánování. Obvyklým řešením plánování je využití standardního plánovače operačního systému nebo JRE. Projekty ANN a Smart Packets využívají navíc vlastní plánovač, který běžící vlákna dodatečně plánuje nastavením priority a přebírá tak rozhodování za standardní plánovač.

Z hlediska vykonávání aplikací se všechny zkoumané implementace spoléhají na vlákna operačního systému nebo JRE. Žádný projekt nepoužívá vlastní implementaci vláken.

4 Plánování

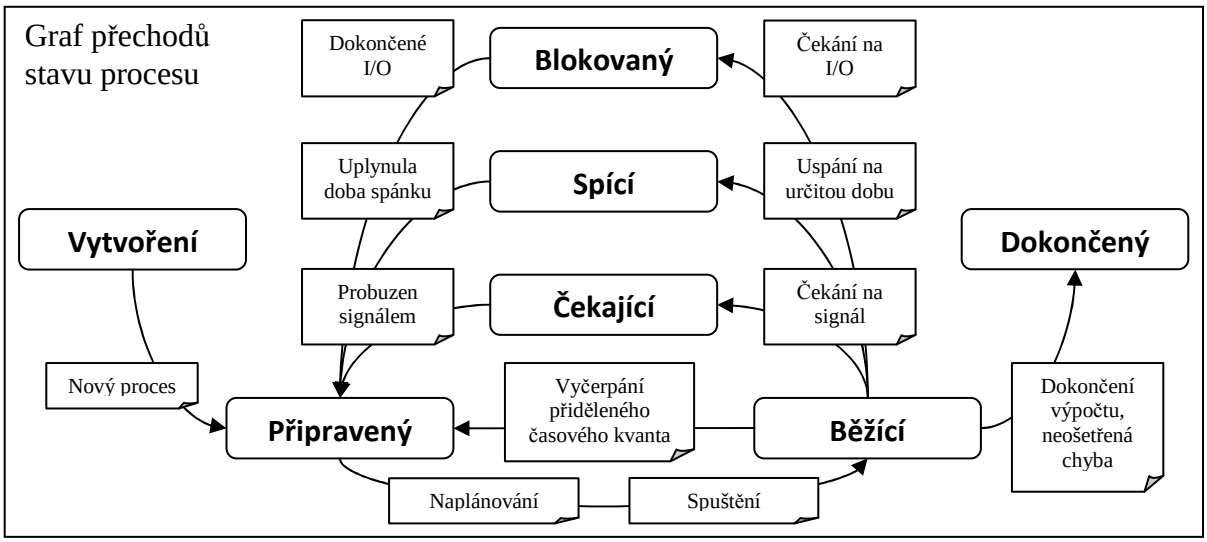
4.1 Proces

Uživatelský program se v terminologii plánování nazývá proces nebo vlákno. Jedná se o program, který je potřeba operačním systémem spustit a vykonat. K vykonání procesu je potřeba nejdříve přidělit systémové prostředky, obvykle procesor a paměť.

Každý proces se může nacházet v jednom z několika definovaných stavů. Tyto stavy pomáhají operačnímu systému určit, jak s procesem manipulovat. Stav procesu může být jeden z následujících:

- Nový – Nově vytvořený proces, který je předán plánovači.
- Připravený – Proces čekající na přidělení procesoru plánovačem.
- Běžící – Proces právě vykonávaný procesorem.
- Čekající – Proces čekající na synchronizačním prostředku (například zámek, monitor, semafor nebo podmíněná proměnná).
- Spící – Proces uspaný na určitou dobu.
- Blokovaný – Proces blokovaný vstupní nebo výstupní operací (například čtení dat ze souboru).
- Dokončený – Proces, který dokončil svůj výpočet nebo skončil neošetřenou výjimkou.

Proces může v závislosti na rozhodnutí plánovače a svojí činnosti měnit svůj stav. Jednotlivé přechody jsou znázorněny na grafu přechodů stavu procesu (*Obr. 01*).



Obr. 01 – Graf přechodů stavu procesu.

V předchozím grafu přechodů uvažujeme, že konec procesu může nastat jen z rozhodnutí samotného procesu nebo v případě neošetřené chyby při výpočtu. V reálném operačním systému může plánovač provádět násilné ukončení procesu z libovolného stavu, například při detekci uvíznutí a jeho následném řešením.

4.2 Definice plánování

Operační systém má za úkol zajistit spuštění a běh uživatelských a systémových procesů (aplikací). Každý počítač má však omezené systémové zdroje, proto musí operační systém zajistit jejich sdílení. Například počet procesorů je vždy omezen, ale počet současně běžících uživatelských procesů bývá obvykle větší. Proto je operační systém vybaven komponentou plánovače, který má na starost přidělování procesoru procesům [SGG00].

Obecně můžeme plánování v operačním systému rozdělit do tří kategorií:

- Dlouhodobé plánování (long-term scheduling) – Plánování spouštění úloh tak, aby byl maximálně využit výpočetní potenciál procesoru. Toho lze docílit například vhodnou kombinací úloh náročných na výpočet (CPU) a úloh náročných na vstup a výstup (I/O). Používá se v dávkových operačních systémech a systémech reálného času.
- Střednědobé plánování (mid-term scheduling) – Plánování odsunu blokovaných, uspaných nebo příliš paměťové náročných procesů z operační paměti, pokud je této paměti nedostatek. Anglický termín pro odsun a návrat procesu v tomto plánování je swapping out a swapping in. Používá se v operačních systémech využívajících virtuální paměť.
- Krátkodobé plánování (short-term scheduling) – Plánování procesoru pro jeho přidělení jednotlivým procesům. Výběr je prováděn ze seznamu pro běh připravených procesů za pomoci zvoleného algoritmu plánování.

Plánování procesů a jejich následné přidělení procesorům může být operačním systémem provedeno, pokud nastane jedna z následujících podmínek:

1) Proces přejde ze stavu běžící do stavu čekající (proces čeká na uvolnění zámku nebo monitoru).
2) Proces přejde ze stavu běžící do stavu spící (proces se sám na určitý čas uspí).
3) Proces přejde ze stavu běžící do stavu blokovaný (proces čeká na dokončení I/O operace).
4) Proces přejde do stavu dokončený (dokončení výpočtu nebo neošetřená chyba).
5) Proces přejde ze stavů čekající, spící, blokovaný do stavu připravený.
6) Proces během svého běhu překročí plánovačem stanovený časový interval (časové kvantum).

V závislosti na používání předchozích podmínek můžeme plánovače dále dělit do následujících tří skupin:

- Nepreemptivní plánování – Plánování nastává jen při naplnění podmínek 1), 2), 3) a 4), ostatní podmínky nejsou použity. Toto plánování se obvykle používá při dávkovém zpracování úloh, operační systém po spuštění procesu musí s dalším přeplánování vyčkat, dokud se proces nevzdá procesoru.

- Preemptivní plánování – Plánování nastává při naplnění libovolné z výše uvedených podmínek. Toto plánování předpokládá úplnou kontrolu nad procesem, proces přidělený procesoru je možné kdykoliv přerušit a později spustit. Základní myšlenkou je přidělování časového kvanta běžícímu procesu. Po vypršení tohoto kvanta je proces za pomoci podmínky 6) pozastaven a je naplánován jiný proces. Ostatní podmínky také platí.
- Plánování reálného času – Speciální plánování pro zajištění deterministické odezvy systému, u každého procesu je nutné splnit dané časové limity na jeho dokončení. Plánování reálného času lze rozdělit na tvrdé plánování (hard real time scheduling), kde je nutné vždy dodržet dané časové limity, protože v případě neúspěchu dojde k chybě, a na měkké plánování (soft real time scheduling), kde nedodržení limitů nezpůsobí chybu, ale mělo by k němu docházet co nejméně.

Zjednodušeně můžeme říci, že nepreemptivní plánová znamená, že proces po získání procesoru jej může využívat po neomezenou dobu na úkor ostatních procesů. Jen proces sám se může vzdát procesoru. Tento způsob je obvykle využíván v dávkových systémech.

V preemptivním plánování naopak proces získává procesor jen na omezenou dobu (časové kvantum), poté je procesu procesor odebrán a plánovačem je přidělen jinému procesu. Tento způsob poskytuje větší kontrolu nad plánováním.

Existuje ještě jeden druh plánování – kooperativní plánování (co-operative scheduling). Jedná se o mezistupeň mezi nepreemptivním a preemptivním plánováním. V základu se jedná o nepreemptivní plánování, tedy naplánovanému procesu je přidělen procesor na libovolně dlouhou dobu. Avšak všechny procesy se v určitých periodách vzdávají procesoru a plánovač může naplánovat jiný. Takto je umožněn souběžný běh více procesů naráz.

Úkolem plánovače je vybrat podle daného algoritmu další proces, kterému bude přidělen procesor. Plánovač je spuštěn vždy, když je splněna jedna z výše uvedených podmínek platných pro daný typ plánovače.

4.3 Kritéria plánovače

Cílem každého plánovače je nejlepší možné dosažení následujících kritérií:

- Vytížení procesoru (CPU utilization) – Plánovač by se měl snažit o nejvyšší možné vytížení procesoru.
- Propustnost (throughput) – Počet procesů, které jsou schopny dokončit svůj běh za jednotku času. Plánovač by se měl snažit o co nejvyšší možnou propustnost.

- Odezva (response time) – Rychlost s jakou plánovač reaguje na změny (zablokování nebo příchod nového procesu). Cílem plánovače je co nejrychleji reagovat a provést přeplánování procesů.
- Doba čekání (waiting time) – Doba, po kterou průměrně proces čeká ve frontě připravených procesů, než je mu přidělen procesor. Plánovač by měl zajistit pro čekající procesy co nejdříve přidělení procesoru.
- Doba jednoho cyklu (turnaround time) – Průměrná doba, po kterou je procesu v plánovacím cyklu přidělen procesor.
- Spravedlivost – Procesor je přidělován procesům spravedlivě podle nějakého kritéria, například pomocí zadané priority procesu nebo podle doby využívání procesoru v nedávné době.
- Výpočetní složitost (computational complexity) – Závislost doby výběru dalšího procesu na počtu plánovatelných procesů. Jedná se o dobu, po kterou se plánovač rozhoduje, kterému procesu bude v následujícím cyklu přidělen procesor. Obvykle se zapisuje pomocí O notace. Cílem je zajistit tuto dobu co nejkratší.

Výše uvedená kritéria mohou být navzájem v konfliktu, proto je nutné mezi nimi nalézt rozumný kompromis pro dosažení vyváženého výkonu.

4.4 Plánovací algoritmy

Výběr dalšího procesu, kterému bude přidělen procesor, se nazývá plánování. Výběr je realizován pomocí plánovacího algoritmu. Algoritmy se mohou navzájem lišit svými vlastnostmi a mírou plnění výše uvedených kritérií.

4.4.1 Nejkratší zbývající čas

Tento algoritmus se označuje zkratkou SJF (shortest job first) nebo SRT (shortest remaining time). Základem tohoto algoritmu je předpoklad, že pro každý proces známe čas, který potřebuje k dokončení výpočtu. Plánovač vybírá vždy proces, jehož čas k dokončení je nejkratší. Pokud má více procesů tyto časy stejné, je zvolen libovolný z nich.

Výpočetní složitost tohoto plánování je závislá na způsobu určení zbývajícího času jednotlivých procesů.

Zbývají čas výpočtu procesů je nutno odhadovat nebo jej procesy musí sami poskytovat. Nutnost odhadovat čas běhu procesů je největší nevýhodou tohoto algoritmu, v případě nesprávného odhadu hrozí snížení propustnosti systému.

4.4.2 Podle pořadí přístupu

Základním algoritmem je plánování podle pořadí přístupu. Obvykle se označuje zkratkou FCFS (first come, first served) nebo FIFO (first in, first out). Každý nový proces je vložen na konec fronty. Plánovač přiděluje procesor vždy procesu na začátku fronty. Pokud

proces přejde do blokovaného stavu, plánovač jej zařadí na konec fronty. Tento algoritmus lze použít jen pro nepreemptivní plánování.

Implementaci tohoto algoritmu je možné zajistit pomocí FIFO fronty. Výpočetní složitost naplánování nového procesu v závislosti na počtu plánovatelných procesů je konstantní O(1). Dochází jen k vyjmutí procesu z čela fronty.

Nevýhodou tohoto algoritmu je obecně horší propustnost, protože algoritmus nedokáže rozlišit mezi výpočetně náročnými a nenáročnými procesy.

4.4.3 Round Robin

Plánovací algoritmus Round Robin (RR) vznikl vylepšením algoritmu podle pořadí přístupu (FCFS). Stejně, jako v případě FCFS algoritmu, i zde jsou procesy řazeny do FIFO fronty. Plánovač po vypršení časového kvanta běžící proces pozastaví a vloží jej na konec fronty. Díky tomuto jednoduchému rozšíření lze zajistit preemptivní plánování.

Implementaci tohoto algoritmu je možné opět zajistit pomocí FIFO fronty. Výpočetní složitost v závislosti na počtu procesů je zde stejná jako v případě FCFS, tedy konstantní O(1).

Nevýhoda tohoto algoritmu je stejná jako v případě FCFS, tedy obecně horší propustnost.

4.4.4 Víceúrovňová fronta

Algoritmus víceúrovňové fronty je značen zkratkou MLQ (multi-level queue). Základem tohoto algoritmu je statická priorita, která je přiřazena procesům. Algoritmus následně využívá tolik front, kolik je různých prioritních tříd. Procesy jsou pak v závislosti na své prioritě přiřazeny do odpovídající fronty. Plánovač následně vybírá a přiřazuje procesor procesům z fronty s nejvyšší prioritou. Pokud proces vyčerpá své časové kvantum, je vrácen na konec fronty. Pokud je fronta prázdná, prohledává plánovač frontu s nižší prioritou. Každé další plánování se provádí opět od fronty s nejvyšší prioritou.

Implementaci tohoto algoritmu je možné zajistit pomocí daného počtu prioritních tříd a odpovídajícího počtu FIFO front. Prioritní třída nemusí být jen konkrétní hodnota priority, může se také jednat o interval priorit. Výpočetní složitost v závislosti na počtu procesů je zde stále O(1), ale je již lineárně závislá na počtu prioritních tříd.

Na rozdíl od RR plánování, MLQ umožňuje plánovat s určitou dávkou priority a upřednostnit tak procesy s vyšší prioritou. Priorita ale nezohledňuje různé chování procesů během vykonávání a prioritu je nutné nastavovat manuálně.

4.4.5 Víceúrovňová fronta se zpětnou vazbou

Algoritmus víceúrovňové fronty se zpětnou vazbou je značen zkratkou MLFQ (multi-level feedback queue). Tento algoritmu vznikl vylepšením algoritmu MLQ, oproti kterému

navíc umožňuje dynamické nastavení prioritní třídy procesu v závislosti na spotřebě časového kvanta. Stejně jako algoritmus MLQ, i zde je využito více front a každá z nich odpovídá jedné prioritní třídě. Proces, který opakovaně spotřebuje celé své časové kvantum, je přesunut do fronty pro nižší prioritní třídu. Naopak proces, který opakovaně nespotřebuje celé své časové kvantum, je přesunut do fronty pro vyšší prioritní třídu. Výběr procesu pro přidělení procesoru je stejný, jako v případě MLQ.

Implementace je téměř totožná jako MLQ, pouze je nutné vkládat proces do fronty podle výše popsaného algoritmu. Výpočetní složitost je shodná s algoritmem MLQ.

Tento algoritmus již umožňuje dynamické plánování, které je plně odvozené od chování procesů. Pokud proces vyžaduje mnoho výpočetního času, je přeřazen do fronty s nižší prioritou a je plánován s menší prioritou. Naopak proces, který příliš nevyžaduje výpočetní čas, zůstává ve vyšší prioritní frontě a tím má zajištěno častější naplánování.

4.4.6 Prioritní plánování

Algoritmus prioritního plánování (priority scheduling) umožňuje také dynamické plánování na základě chování svých procesů. Každý proces má přiřazenu počáteční prioritu, která se bude měnit v závislosti na jeho chování. Základní myšlenkou je posilování priority procesů, které čekají na naplánování (nebo naopak oslabování priority běžících procesů). Výběr procesu pro přidělení procesoru je následně proveden nalezením procesu s nejvyšší prioritou.

Určení konkrétní priority může být zajištěno například pomocí velikosti naposledy spotřebovaného časového kvanta. Plánovač si pro každý proces uloží tuto hodnotu a čekajícím procesům ji bude v každém cyklu například zmenšovat o polovinu. Naopak běžícím procesům bude tato hodnota s každým dalším cyklem jen narůstat.

K implementaci tohoto algoritmu postačuje seznam procesů. Výpočetní složitost v závislosti na počtu procesů je lineární O(n), neboť je při každém plánování potřeba mezi všemi procesy nalézt proces s nejvyšší prioritou. Výběr procesu je možné urychlit na O(log n) ukládáním procesů do stromu, jako klíč bude použita doba naposledy spotřebovaného časového kvanta.

Výhodou tohoto algoritmu je, stejně jako v případě MLFQ, plně dynamické plánování odvozené od chování procesů.

4.4.7 Plánování loterií

Plánování loterií je speciální algoritmus založený na náhodě a pravděpodobnosti. Každý proces obdrží určitý počet tiketů. Čím má proces nastavenou vyšší prioritní třídu, tím více tiketů obdrží. Plánovač při každém cyklu náhodně losuje a naplánuje proces, který vylosovaný tiket vlastní.

Problémem tohoto algoritmu je implementace tiketů a jejich přiřazení procesům, je nutné tuto implementaci zajistit efektivně. Také je nutné použít generátor náhodných čísel rovnoměrného rozdělení pravděpodobnosti.

4.5 Plánování pro více procesorů

Všechny výše uvedené algoritmy vždy vycházejí z předpokladu, že je k dispozici jen jeden procesor. V dnešní době jsou již běžně dostupně víceprocesorové počítače. Je tedy vhodné algoritmy upravit tak, aby dokázaly využívat potenciálu více procesorů.

V dalších úvahách předpokládejme symetrický multiprocesor (SMP) o *n* procesorech.

Plánovací algoritmy vždy vybírají jen jeden proces pro přidělení procesoru, avšak vždy tak činí na základě určitého kritéria. Prioritní plánování vybírá vždy proces s nejvyšší prioritou, SJF vybírá proces s nejnižším časem do ukončení, FCFS, RR, MLF a MLFQ vybírají proces z čela odpovídající fronty. Kritérium je možné rozšířit a vybírat ne jeden proces, ale *n* procesů s nejvyšší prioritou (prioritní plánování), nejnižším časem do ukončení (SJF) nebo z odpovídající fronty jednoduše vybrat ne jeden, ale *n* procesů.

Dále je vhodné zajistit určitou optimalizaci přidělení procesů na procesory. Pokud bude plánovač proces střídavě plánovat na různé procesory, nevyužije se zcela potenciál vyrovnávacích cache pamětí v procesorech. Proto je vhodné plánování procesů na procesory neprovádět libovolně, ale snažit se ho určitým způsobem optimalizovat.

Předpokládejme *n* procesorů, na kterých může běžet nejvýše *n* různých procesů. Po uběhnutí časového kvanta plánovač běžící procesy pozastavil (přesunul mezi připravené procesy) a vybral novou podmnožinu připravených procesů velikosti nejvýše *n*. V této podmnožině se mohou nacházet také procesy, které před přeplánováním již běžely. Tyto procesy by měly být přiřazeny na procesory pomocí posloupnosti následujících pravidel:

1) Pokud byl vybraný proces naplánován i v minulém cyklu (proces nyní běží na některém z procesorů), je naplánován znovu na stejný procesor.
2) Pokud byl vybraný proces naposledy vykonáván na procesoru, který je nyní volný, je proces na tento procesor naplánován.
3) Vybraný proces je naplánován na libovolný volný procesor.

Předchozí pravidla je nutné aplikovat postupně. Na každý vybraný proces lze aplikovat právě jedno pravidlo. Na všechny plánovačem vybrané procesy je aplikováno pravidlo 1). Na procesy, které nesplnily pravidlo 1), je aplikováno pravidlo 2). Pravidlo 3) je aplikováno na všechny zbylé procesy, které nesplnily kritéria předchozích pravidel 1) a 2).

V reálné implementaci nebudou procesy pozastaveny a odebrány procesorům před přeplánováním, ale až na základě výsledku plánování. Například při aplikaci pravidla 1) není potřeba proces pozastavovat a spouštět, neboť již na žádaném procesoru běží.

Tato pravidla přiřazení nemusí být přímo součástí plánovacího algoritmu, mohou být přítomna jako mezivrstva, která jen upraví výstup plánovače (přiřadí zvolené procesy nejvhodnějším procesorům). Díky tomu není potřeba zasahovat do algoritmu plánovačů. Postačuje je upravit tak, aby vracely nejvýše ne jeden, ale *n* procesů pro naplánování.

4.6 Zavaděč

Zavaděč (dispatcher) je součást plánovače přistupující k procesoru. Jeho úkolem je spouštění a pozastavování procesů podle rozhodnutí plánovacího algoritmu.

4.7 Fiber vlákna

Fiber vlákno je vlákno zcela implementované v uživatelském prostoru. Obvykle fiber vlákna svým aplikacím poskytují virtuální stroje, protože je vyžadována úplná kontrola nad spuštěnou aplikací a odstínění aplikace od operačního systému, ve kterém běží samotný virtuální stroj.

Výhodou této implementace je spouštění, vykonávání a přepínání fiber vláken v uživatelském prostoru bez nutnosti zásahu jádra operačního systému a jeho plánovače, tedy omezení systémových volání. Další výhodou může být absolutní kontrola nad prováděným kódem.

Nevýhodou je však nutnost vlastní implementace plánovače a potřebných synchronizačních prostředků. Použití synchronizačních prostředků operačních systémů není možné, neboť by nezablokovaly jen fiber vlákno, ale také interpretr, který vlákno vykonává.

4.8 Uvíznutí

Uvíznutí (deadlock) je situace, která je způsobena vzájemným čekáním více vláken. Vlákna z určité množiny čekají na různé akce, které však mohou provést opět jen vlákna z této množiny [SGG00].

V dalších úvahách budeme uvažovat jen uvíznutí při zabírání zámků. Vlákna se pokoušejí zabírat zámky. Pokud se vlákno pokusí zabrat zámek, který již vlastní někdo jiný, musí vlákno počkat na uvolnění zámku. Během čekání nemůže vlákno provádět žádnou činnost.

4.8.1 Definice uvíznutí

K uvíznutí vláken může dojít, pokud jsou splněny všechny čtyři následující podmínky:

1) Vzájemné vyloučení – Každý zámek smí být vlastněn nejvýše jedním vláknem. Zámek nelze sdílet mezi více vláken.
2) Vícenásobné zabírání zdrojů – Vlákno může zabrat více zámků. Při každém zabírání může vlákno čekat na uvolnění zabíraného zámku.
3) Neodnímatelnost – Vláknu nelze odebrat zámek, dokud se ho nevzdá.
4) Cyklické čekání – Vlákna mohou čekat na uvolnění zámku vlastněného jiným vláknem v kruhu.

Pokud se podaří zajistit nesplnění alespoň jedné z předchozích podmínek, nemůže k uvíznutí dojít.

4.8.2 Detekce uvíznutí

Detekce uvíznutí může být provedena pomocí grafu alokace zdrojů (RAG – resource allocation graph). Jedná se o orientovaný graf. Uzly grafu tvoří vlákna a zámky. Hrany grafu určují vztahy. Orientovaná hrana z uzlu zámku do uzlu vlákna značí zabrání zámku vláknem. Orientovaná hrana z uzlu vlákna do uzlu zámku značí čekání vlákna na uvolnění zámku.

Po sestavení je graf prohledán vyhledáváním do šířky (BFS – breadth-first search). Cílem je nalézt cyklus, který znamená uvíznutí. Pokud je cyklus nalezen, je jedno vlákno ležící na cestě cyklu násilím ukončeno a zámek je mu odebrán. Vybráno je vlákno, které vlastní nejméně zabraných zámků. Pokud je více různých vláken se stejným počtem zabraných zámků, je vybráno libovolné z nich.

Pokud je v grafu více cyklů, je vyhledávání spouštěno opakovaně, dokud se všechny cykly nepodaří odstranit.

4.8.3 Prevence uvíznutí

Prevenci uvíznutí je možné zajistit pomocí bankéřova algoritmu. Bankéř spravuje veškeré zdroje (zámky) a rozhoduje o jejich přidělení. Při přidělování každého zdroje si bankéř vždy ověřuje, zda se nemůže dostat do nebezpečného stavu.

Bezpečný stav je takový stav, pro který existuje alespoň jedna posloupnost přidělení zdrojů, která povede k uspokojení potřeb všech vláken.

Opakem je nebezpečný stav, ve kterém již neexistuje žádná cesta k naplnění potřeb všech zúčastněných vláken. Nebezpečný stav neznamená okamžité uvíznutí. Znamená ale situaci, která v budoucnu nevyhnutelně povede k uvíznutí.

Pokud bankéř zjistí, že přidělením zdroje dojde k přechodu z bezpečného stavu do nebezpečného, přidělení zdroje nepovolí a žadatel je blokován. Zdroj je žadateli přidělen až v situaci, kdy jeho přidělení nepovede do nebezpečného stavu.

Nevýhodou tohoto algoritmu je potřeba znát dopředu požadavky všech žadatelů. V našem případě je potřeba znát posloupnost zámků, které vlákna pro své dokončení potřebují získat.

5 SAN – Smart Active Node

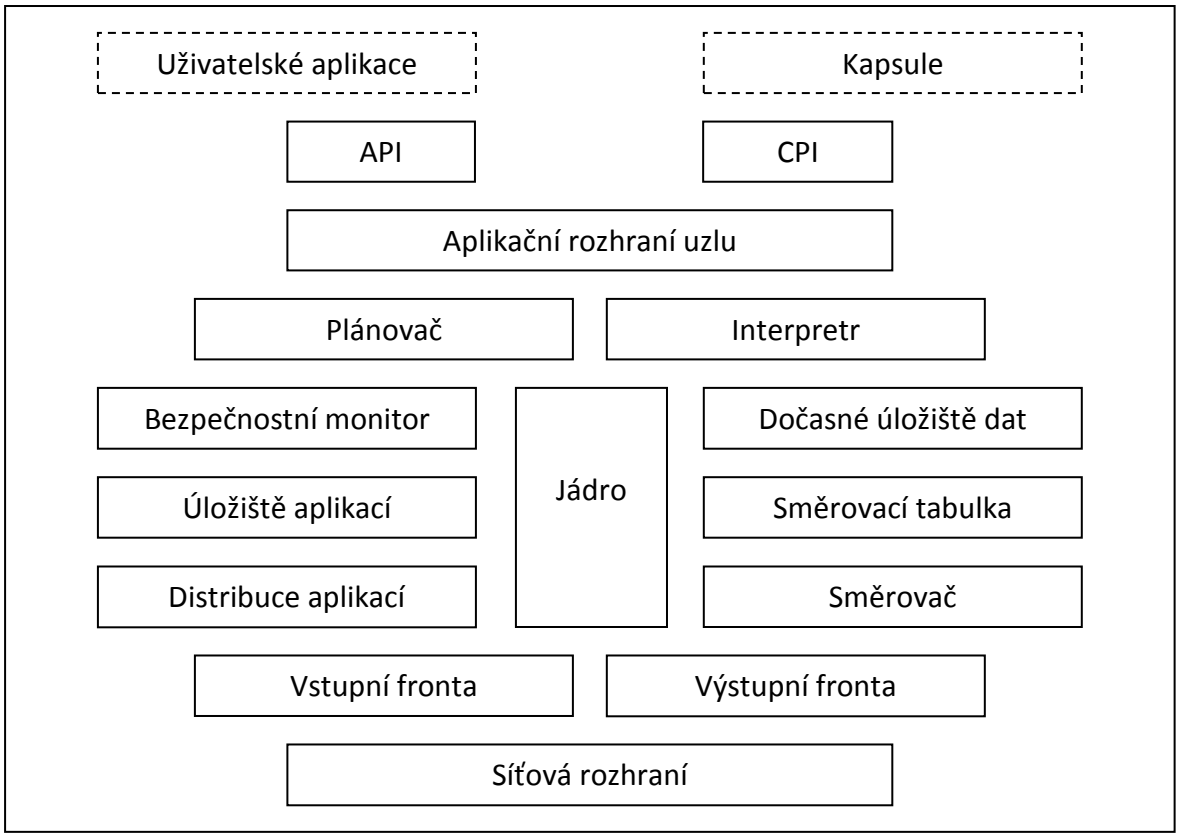
Projekt SAN vznikl na Západočeské univerzitě v Plzni v roce 2007. Jedná se o projekt zabývající se aktivními sítěmi [Rej08]. Předchůdcem tohoto projektu byl projekt Grade32.

5.1 Předchůdce – Grade32

V roce 2004 vznikl na Západočeské univerzitě v Plzni projekt Grade32 zabývající se aktivními sítěmi [Kou04]. Jeho cílem bylo vytvořit distribuované výpočetní prostředí s možností dynamického přerozdělování zátěže. Směrování tohoto projektu na distribuované výpočty bohužel znemožnilo zajistit některé požadované vlastnosti aktivních sítí, například bezpečnost a ověřování uživatelských aplikací. V původním projektu se nepočítalo s nasazením do otevřené veřejné sítě, kde uživatelé budou spouštět své vlastní a potencionálně nebezpečné aplikace. Nástupcem projektu Grade32 se stal projekt aktivní sítě SAN, jehož cílem je odstranit výše uvedené nedostatky.

5.2 Architektura uzlu

Uzel SAN se skládá z několika komponent a modulů uspořádaných ve vrstvách. Detailní pohled na strukturu uzlu poskytuje následující obrázek (*Obr. 02*).



Obr. 02 – Architektura uzlu SAN.

5.3 Popis jednotlivých komponent a vrstev

- Síťová rozhraní – Tato vrstva se stará o přístup ke komunikačnímu médiu (TCP/IP nebo UDP/IP síť) a zajišťuje komunikaci mezi jednotlivými uzly. Zároveň tato vrstva

odstiňuje vyšším vrstvám konkrétní síťové technologie pro přístup ke komunikačnímu médiu a poskytuje abstrakci službami, které pracují na úrovni kapsulí (zasílání a přijímání).

- Vstupní a výstupní fronta – Tyto fronty se starají o vyrovnávací paměť pro přijaté nebo odeslané kapsule. Umožňují kapsule odložit a zajistit tak asynchronnost mezi jednotlivými vrstvami, které nemusejí čekat na dokončení zpracování kapsule. Toto umožňuje paralelní práci jednotlivých vrstev.
- Jádro – Základní komponenta starající se o zavedení, konfiguraci a spuštění ostatních komponent.
- Úložiště aplikací – Komponenta starající se o perzistentní ukládání a správu kódu aplikací.
- Distribuce aplikací – Komponenta zajišťující odesílání kódu aplikací ostatním uzlům a zároveň získávání kódu od ostatních uzlů.
- Směrovací tabulka – Struktura pro uchovávání aktuálního stavu směrování.
- Směrovač – Komponenta pro správu směrovací tabulky. Směrovač zajišťuje odesílání kapsulí na správné síťové rozhraní tak, aby kapsule dosáhly svého cílového uzlu.
- Dočasné úložiště – Struktura pro dočasné ukládání uživatelských dat ve formě Java objektů. Jedná se o implementaci synchronizačního prostředku Blackboard [EMD88]. Každá položka má časově omezenou dobu existence. Tuto součást uzlu lze přirovnat ke cookies v internetovém prohlížeči. Uložená data nejsou perzistentní, po ukončení uzlu jsou všechna ztracena.
- Bezpečnostní monitor – Komponenta zajišťující bezpečnost na uzlu pomocí uživatelských účtů. Ke každému účtu jsou přiřazena přístupová práva k určitým úkonům a částem aplikačního rozhraní uzlu, které smí uživatel nebo aplikace používat. Během interpretace kódu tato komponenta schvaluje volání veškerých metod, a to nejen metod aplikačního rozhraní uzlu, ale také těch ze standardní knihovny JRE. Dalším úkolem bezpečnostního monitoru je sledování systémových zdrojů, například množství alokované paměti a doba, po kterou aplikace využívají procesor.
- Plánovač – Komponenta přidělující jednotlivé aplikace a kapsule interpretru, který je vykonává. Kromě přidělování procesorového času zajišťuje také plánovač vyžádání chybějícího aplikačního kódu a omezuje počet najednou běžících kapsulí.
- Interpretr – Komponenta starající se o vykonávání uživatelských aplikací.
- Aplikační rozhraní uzlu – Soubor metod uspořádaných do rozhraní, které využívají aplikace a kapsule pro ovládání uzlu. Patří sem například spouštění nových vláken, čekání na jejich ukončení, odesílání nových kapsulí nebo ovládání směrování. Aplikační rozhraní uzlu je přístupné aplikacím pod názvem API (Application Programming Interface) a kapsulím pod názvem CPI (Capsule Programming Interface). API a CPI se liší v dostupných metodách, obecně je API a CPI podmnožina celkového aplikačního rozhraní uzlu.

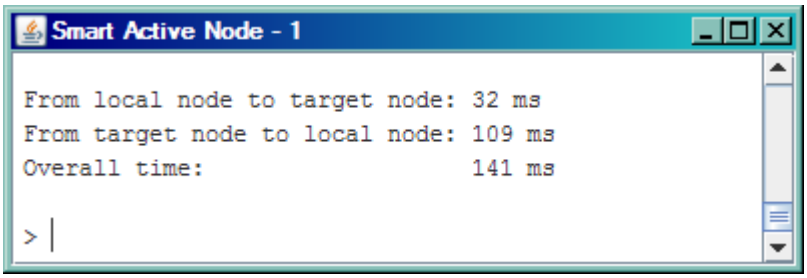
5.4 Platforma

Projekt SAN je celý napsaný v jazyce Java verze 1.6. Uzel SAN je proto možné spustit na libovolném operačním systému, pro který existuje běhové prostředí JRE. Lze jej spustit na všech obvyklých distribucích operačních systémů Linux a Windows.

Pro sestavení, spouštění a přípravu uživatelských aplikací je využíván sestavovací nástroj Ant.

Logování provozu a ladících výpisů zajišťují nástroje knihovny Java, knihovna Log4j a vlastní implementace logovacího nástroje uzpůsobená potřebám uzlu SAN.

Uživatelské rozhraní je zajištěno pomocí textové konzole nebo pomocí vlastní grafické konzole vytvořené v JFC Swing. Vzhled grafické konzole je znázorněn na následujícím obrázku (*Obr. 03).*



Obr. 03 – Grafická konzole SAN.

Existuje také verze uzlu SAN napsaná v jazyce C/C++. Tato verze vznikla jako průběžný přepis (port) z jazyka Java. Účelem této verze bylo zvýšení výkonu, který je dosažen právě přepsáním do jazyka C/C++, který umožňuje kompilaci kódu přímo do instrukční sady procesoru. V této práci se ale touto verzí nebudeme zabývat, neboť vývoj plánování projektu SAN probíhá ve verzi v jazyce Java.

Do budoucna se plánuje vývoj projektu SAN++ (nová verze SAN od základu v C/C++), kde budou využity i poznatky z této diplomové práce.

5.5 Konfigurace uzlu

Uzel SAN vyžaduje pro své spuštění konfigurační XML soubor. V tomto souboru je uložena výchozí konfigurace směrovací tabulky virtuální překryvné sítě, jednotlivá síťová rozhraní, jejich virtuální adresy a reálné IP adresy a porty. Dále je zde uloženo jméno uzlu, typ uživatelského rozhraní uzlu (textová nebo grafická konzole), typ a úroveň logování provozu uzlu, adresář s úložištěm aplikací a několik dalších nastavení.

5.6 Uživatelské aplikace

Uživatelské aplikace a kapsule jsou programy napsané v jazyce Java, které se spouštějí na uzlech SAN. Kapsule je spuštěna, když dorazí na uzel. Aplikace je nutné spouštět ručně

z ovládací konzole uzlu. Jednotlivé aplikace jsou uloženy jako Java bytecode a z tohoto bytecode je spočítán kontrolní součet (hash), který slouží jako identifikátor aplikace.

Aplikace musí ve svém programovém balíku specifikovat, pro jaký interpretr Javy je určena. V současné době je k dispozici Bean Shell interpretr a SAN interpretr (vyvinutý pro uzel SAN).

Původně měl interpretaci zajišťovat jen Bean Shell interpretr, avšak malá výkonnost a nemožnost sledovat a ovlivňovat samotnou interpretaci vedla k vývoji vlastního SAN interpretru, který je nyní upřednostňován. Přesto je však Bean Shell interpretr stále v projektu SAN přítomen a využívá se pro spuštění aplikace uživatelského rozhraní, kterou SAN interpretr v současné verzi spustit nedokáže. Uživatelské rozhraní, textová a grafická konzole jsou v uzlu SAN zakomponovány jako uživatelská aplikace na nejvyšší úrovni, která jako své potomky spouští ostatní aplikace a její ukončení znamená ukončení činnosti celého uzlu.

Počítáním kontrolního součtu z bytecode je zajištěno, že při jakékoliv změně kódu vznikne nový jedinečný identifikátor. Díky tomuto je každá aplikace a její konkrétní verze v celé síti SAN jedinečná a nemůže dojít k použití nesprávného kódu k vykonávání kapsulí.

Pro komunikaci mezi aplikací a SAN uzlem slouží aplikační rozhraní, které uzel poskytuje běžícím aplikacím a kapsulím. Jedná se o soubor metod uspořádaných do rozhraní, které je aplikaci při startu předáno a které následně aplikace dle potřeby volají.

Aplikace jsou kompilovány pomocí standardního kompilačního nástroje `javac` z JDK, ale interpretovány jsou již vlastním interpretrem Java-In-Java (SAN interpretr), který je součástí každého uzlu. Kompilací nástrojem `javac` z JDK je navíc zajištěna statická optimalizace kódu.

5.7 Virtuální síť

Aktivní síť SAN využívá pro svojí komunikaci vlastní překryvnou síť nad TCP/IP sítí. Každý uzel SAN má v této síti přiřazenou jedinečnou adresu. Dále je možné tuto virtuální síť rozdělit na menší podsítě a adresování usnadnit pomocí identifikátorů těchto podsítí.

5.8 Směrování

Pro zajištění směrování je každý uzel SAN vybaven vlastním směrovačem a směrovací tabulkou, jejíž základní konfigurace se provádí pomocí konfiguračního XML souboru. Konfiguraci směrovací tabulky je možné upravovat i za běhu pomocí uživatelského rozhraní uzlu.

5.9 Distribuce aplikačního kódu

Kapsule cestující aktivní sítí si s sebou nese jen uživatelská data a identifikátor (kontrolní součet) svého aplikačního kódu. Když je potřeba doručenou kapsuli na uzlu spustit, je nejdříve přečten její identifikátor a prohledáno úložiště kódu. Pokud není požadovaný

aplikační kód nalezen, je třeba si jej vyžádat od ostatních uzlů. Uzel SAN podporuje různé strategie distribuce aplikačního kódu aplikací [Ste09]. Nejjednodušší strategií je začít uzlem, který kapsuli odeslal, protože tento uzel určitě aplikační kód kapsule vlastní. Po získání aplikačního kódu je možné vykonat uživatelskou akci, ke které je kapsule určena.

5.10 Kapsule

Základní jednotkou komunikace mezi jednotlivými uzly SAN je kapsule. Kapsule je pro aktivní síť tím, čím je paket pro TCP/IP síť. Všechny vrstvy (s výjimkou té nejnižší vrstvy) architektury SAN pracují a komunikují jen pomocí kapsulí. Jen nejnižší vrstva se stará o přístup ke komunikačnímu médiu (zde TCP/IP nebo UDP/IP síť) a zajišťuje vzájemnou komunikaci mezi jednotlivými uzly SAN a ostatním vyšším vrstvám poskytuje své služby na úrovni přijímání a zasílání kapsulí a odstiňuje je od komunikačního média.

Pro vložení kapsule do aktivní sítě se používá speciální termín – injektovat. Injektovaná kapsule je následně přeposílána mezi jednotlivými SAN uzly, dokud si sama nevyžádá své odstranění ze sítě.

Kapsule si může sama vybrat, zda bude spouštěna na každém uzlu, který navštíví, nebo bude jen přeposílána a spuštěna až na svém cílovém uzlu. Pokud je kapsule na uzlu spuštěna, může přistupovat ke směrovači uzlu a změnit adresu svého cíle.

Datový rámec kapsule je zobrazen na následujícím obrázku (*Obr. 04*).

ID	DEST	SRC	IDC	IDA	HOP_LIMIT	OPTIONS	DATA

Obr. 04 – Datový rámec kapsule SAN.

Význam jednotlivých polí rámce:

- ID – Konstanta pro identifikaci rámců kapsulí SAN (magic word).
- DEST – Cílová adresa uzlu.
- SRC – Zdrojová adresa uzlu.
- IDC – Identifikace kapsule.
- IDA – Identifikace aplikace, která kapsuli injektovala.
- HOP_LIMIT – Maximální počet navštívených uzlů.
- OPTIONS – Řízení sítě.
- DATA – Uživatelská data přenášená kapsulí.

Kapsule využívané uzly SAN lze rozdělit do 3 základních skupin:

- Uživatelské kapsule – Tyto kapsule odesílají uživatelské aplikace, mohou přenášet data pro tyto aplikace nebo vykonávat uživatelský kód na vzdálených uzlech.
- Kapsule s žádostí o aplikační kód – Tyto kapsule slouží k vyžádání aplikačního kódu uzly, které obdrželi kapsuli, ale nemají pro ni spustitelný kód.

- Kapsle s kódem aplikace – Tyto kapsule slouží k přenosu aplikačního kódu kapsulí mezi jednotlivými uzly.

5.11 Identifikátory

Každé spuštěné aplikaci nebo kapsuli přiřazuje plánovač jedinečný identifikátor (`Guid`). Tento identifikátor slouží například při čekání na ukončení spuštěných vláken nebo při doručení dat z příchozí kapsule běžící aplikaci, která tuto kapsuli vytvořila a vyslala do sítě.

5.12 Procesy

Každá uživatelská aplikace nebo kapsule je spuštěna jako nový proces. Pro umožnění výpočtu ve více vláknech může proces spouštět další procesy.

Reprezentaci procesu zajišťuje třída `san.core.Process`. Zde je uložen veškerý kontext daného procesu.

Proces se může nacházet v jednom z následujících stavů:

- `READY_TO_RUN` – Proces je připravený ke spuštění a čeká na přidělení procesoru.
- `RUNNING` – Proces je spuštěný, byl mu přidělen procesor a právě probíhá výpočet.
- `FINISHED` – Proces dokončil svůj výpočet a skončil.

5.13 Správce interpretrů

K vykonávání aplikací nebo kapsulí je možné používat různé interpretry. Informace o použitém interpretru je uložena v identifikátoru programového kódu aplikace. Přístup k interpretrům je zajištěn pomocí komponenty správce interpretrů, který funguje jako vrstva mezi plánovačem a jednotlivými interpretry. Na požadavek plánovače poskytne přístup k interpretru odpovídajícímu spouštěné aplikaci a naopak upozorňuje plánovač na skončení interpretace některého ze spuštěných procesů.

Další činností správce interpretrů je zajištění doručování dat z příchozí kapsule do běžící aplikace. Samotné doručení dat je zajištěno přímým přístupem do datového prostoru aplikace a jeho následná úprava přímým změněním obsahu zvolených proměnných.

V současné době jsou k dispozici následující interpretry:

- SAN interpretr – Preferovaný interpretr.
- Bean Shell interpretr – Interpretr jen pro aplikaci grafického rozhraní, nepodporuje dostatečnou míru zabezpečení, v porovnání se SAN interpretrem je pomalejší.

5.14 Interpretace kódu

Interpretr Java bytecode se stará o vykonávání uživatelských aplikací. Aplikace a kapsule nelze spouštět přímo pomocí JRE. Je nutné umožnit komponentě bezpečnostního monitoru jejich sledování a schvalování nebo zamítání jejich jednotlivých činností. Nelze tedy použít Bean Shell interpretr, ale je nutné použít interpret vlastní a zcela řídit vykonávání aplikací na úrovni jednotlivých instrukcí [Syr09].

Pokud aplikace nebo kapsule při vykonávání skončí chybou (způsobí neošetřenou výjimku), nebo pokud poruší bezpečnostní pravidla, je její vykonávání ukončeno.

Interpretr nerozlišuje mezi uživatelskou aplikací nebo kapsulí, v obou případech je posloupnost operací potřebných pro spuštění a vykonávání programového kódu stejná. Jediný rozdíl je v aplikačním rozhraní, které je aplikaci nebo kapsuli předáno (API nebo CPI). Jinými slovy řečeno, rozdíl je jen v množině služeb uzlu, které může aplikace nebo kapsule využívat. Například kapsule nemůže spouštěn nová vlákna, na rozdíl od aplikace.

Aplikace může spouštět jen uživatel ručně z uživatelské konzole nebo jiná aplikace jako svého potomka. Kapsule může spouštět jen uzel SAN po jejich přijetí ze sítě.

V terminologii plánování pro projekt SAN lze aplikace nebo kapsule souhrnně popsat termínem proces. Proces je vytvořen plánovačem, který jej předává interpretru k vykonání. Každému procesu je přiřazen jednoznačný identifikátor (`Guid`). Díky tomuto identifikátoru je možné zajistit meziprocesovou komunikaci, například čekání na skončení procesu nebo doručení dat z příchozí kapsule do aplikace.

Interpretry SAN ani Bean Shell neumožňují ukládání stavu právě vykonávaného procesu. V případě interpretru SAN je zásobník volání implementován pomocí rekurze (rekurzivní volání metody `runMethod(…)`).

5.15 Plánování aplikací

Hlavním úkolem plánovače je rozhodovat o spouštění procesů a jejich přidělování interpretru. Plánovač se spouští periodicky v určitém intervalu a provádí výběr nejvhodnějšího procesu určeného k interpretaci.

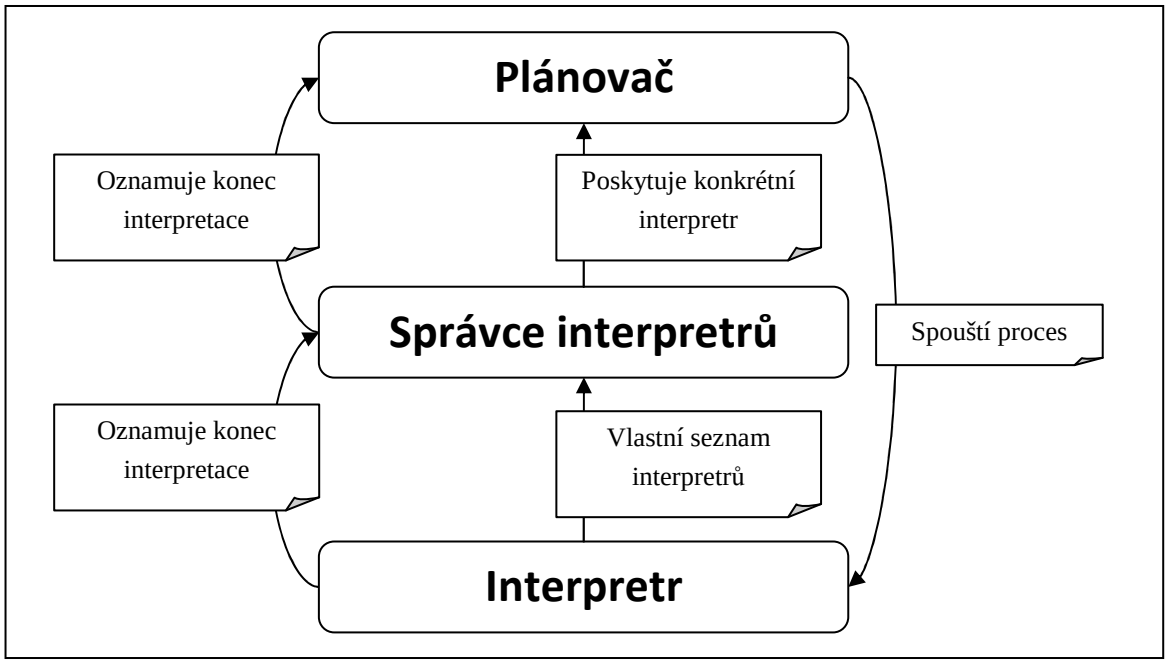
Ačkoliv je pro vlastní plánování připraveno odpovídající rozhraní, plánovač je nevyužívá. Interpretr SAN a Bean Shell je pro každý nový proces spuštěn v novém vlákně JRE, ve kterém je proces následně vykonáván. Tímto je zajištěna možnost paralelního vykonávání více procesů současně. Plánovač zde tedy slouží jen ke spouštění nových procesů.

Dalším úkolem plánovače je zajištění služby čekání na skončení procesu (metoda `waitForApplication(Guid guid)`) a vyžadování programového kódu příchozích kapsulí. Pokud pro kapsuli není kód k dispozici, je kapsule pozdržena, dokud kód komponenta distribuce aplikací pro kapsuli nezíská.

5.16 Práce plánovače a interpretru

Vzájemné propojení plánovače a interpretrů je zajištěno pomocí rozhraní. Plánovač přistupuje k interpretru pomocí správce interpretrů. Správce interpretrů poskytuje konkrétní interpretr pomocí rozhraní `IInterpreter`. Interpretr přistupuje k plánovači prostřednictvím správce interpretrů a rozhraní `IScheduler` poskytovaného plánovačem.

Plánovač vkládá do interpretru procesy určené ke spuštění a interpretr po dokončení vykonávání procesu tuto skutečnost oznamuje plánovači. Tento koloběh je znázorněn na obrázku *Obr. 05*.



Obr. 05 – Vztah plánovače a interpretru.

Rozhraní interpretru obsahuje také metody pro přerušení vykonávání procesu a jeho odebrání z interpretru, tyto metody nyní nejsou využity. Jakmile započne interpretace, je vždy spuštěna v novém vlákně a nelze ji pozastavit nebo přerušit.

5.17 Slabá místa

Současná implementace plánovače a interpretru není vyhovující. Vytvoření zcela nového vlákna je časově náročné, každý proces vyžaduje vlastní instanci interpretru. Nelze určovat prioritu vykonávání jednotlivých procesů, a proto nelze procesy efektivně plánovat. A také nelze zcela využít potenciálu víceprocesorového hardware.

5.17.1 Vytváření nových vláken

Vytváření nových vláken při startu každého procesu zabírá určitou režii. Tato režie může ovlivňovat časově kritické aplikace. Například aplikace ping, která měří rychlost reakční doby uzlů. Výsledek aplikace může ovlivnit právě doba potřebná na zavedení a spuštění nového vlákna.

Další problém může tento přístup ke správě vláken způsobit, pokud je uzel zatížen velkým množstvím kapsulí. Pokud budeme předpokládat, že průměrná kapsule po svém spuštění provede krátký výpočet a následně skončí, a takovýchto průměrných kapsulí bude aktivní sítí procházet mnoho, zjistíme, že uzel SAN bude určitou (zřejmě nezanedbatelnou) část svého výkonu spotřebovávat na neustálé vytváření nových vláken [SK10].

5.17.2 Priorita vláken

Vlákna, ve kterých běží jednotlivé procesy, plánuje přímo virtuální stroj JRE, nikoliv plánovač uzlu SAN, který se nyní stará jen o zavádění procesů. Přestože Java umožňuje nastavit prioritu svých vláken, toto řešení neumožňuje detailnější a dynamické nastavování priority potřebné v projektu SAN. U běžících procesů nelze předem určit, jakým způsobem se budou chovat a jakým způsobem budou využívat systémové prostředky (procesorový čas a alokovanou paměť), je potřeba plánovačem prioritu upravovat neustále v závislosti na tom, jak se jednotlivé procesy chovají.

Aby bylo možné určit, jak prioritu běžících procesů upravovat, je potřeba znát chování těchto procesů, tedy jak moc využívají procesorový čas. To bohužel v případě Java vláken snadno zjistit nelze. K jejich synchronizaci je nutné použít také synchronizační prostředky jazyka Java, nad kterými nemáme z pohledu plánovače žádnou kontrolu. Bylo by možné v krátkých intervalech testovat stav vláken a podle něj určit, zda vlákno běží nebo zda spí či čeká, ale toto řešení by bylo neúplné. Nelze určit, zda vlákno neběží, protože námi vykonávaný proces se skutečně vzdal procesoru a neběží, nebo protože plánovač virtuálního stroje Javy vlákno na okamžik pozastavil z jiných důvodů. A naopak nelze vlákno zcela pozastavit, neboť nemáme úplnou kontrolu nad činností plánovače JRE.

5.17.3 Synchronizační prostředky

Použitím synchronizačních prostředků jazyka Java se při interpretaci připravujeme o kontrolu nad vykonávaným kódem. Aby bylo možné tyto prostředky využít, je nutné k nim přistupovat pomocí reflexe. Reflexe je pokročilý nástroj jazyka Java, který umožňuje za běhu aplikace dynamicky přistupovat k proměnným, volat metody a vytvářet instance tříd. Například služba čekání na ukončení aplikace (`waitForApplication(Guid guid)`) využívá metodu `Object.wait()` volanou pomocí reflexe.

Schopnosti mechanizmu reflexí jsou bohužel vykoupeny jeho náročností. Volání metod rozhraní uzlu SAN pomocí reflexe je výrazně pomalejší, než přímé volání, které v případě interpretru SAN nelze snadno zajistit. Od volání metod očekáváme také předávání a vracení parametrů. Bylo by nutné implementovat každou jednotlivou metodu z aplikačního rozhraní uzlu SAN přímo do interpretru a zajistit náležitou konverzi neprimitivních datových typů, a to v obou směrech mezi jádrem uzlu SAN a prostorem interpretovaného procesu (interpretr používá vlastní struktury pro uchovávání jednotlivých objektů běžícího procesu).

Přesto je nástroj reflexe pro nás v tuto chvíli nenahraditelný. Například přístup aplikace k aplikačnímu rozhraní je zajištěn právě pomocí reflexe a je to zřejmě jediný způsob, jak umožnit stavovém prostoru aplikace přístup k obecně libovolnému rozhraní uzlu SAN. Stejně tak přístup k nativním metodám lze zajistit jen pomocí reflexe. Nativní metody z principu nelze interpretovat, protože k nim neexistuje bytecode a virtuální stroj Javy je obsahuje jen jako platformě závislé knihovny. Přesto bychom se měli snažit používat mechanizmus reflexí opravdu jen tam, kde je nezbytně nutný, a pokud možno v co nejmenší míře.

Není nutné používat synchronizační prostředky Javy volané interpretrem pomocí reflexe. Je možné si na úrovni plánovače vytvořit vlastní synchronizační prostředky, jejichž volání nebude zatížené použitím nástroje reflexe. A navíc je možné tyto vlastní synchronizační prostředky přímo přizpůsobit potřebám aktivních sítí.

5.17.4 Spotřeba paměti

Dalším problémem je potřeba pro každý nový proces vytvářet novou instanci interpretru, čímž se zvyšují paměťové nároky.

5.17.5 Využívání víceprocesorového HW

Pokud uzel SAN běží na víceprocesorovém hardwaru, tak při použití předem neznámého počtu různě se chovajících vláken nelze zajistit optimální vytížení procesoru, tedy jedno počítající vlákno na jeden procesor. Tímto se připravujeme o část potencionálního výkonu.

Plánovač virtuálního stroje jazyka Java (JRE) bude díky režii našeho plánovače vždy efektivnější než jakýkoliv plánovač, který pro uzel SAN dokážeme vytvořit. Bohužel protože není možné přizpůsobit plánovač JRE, aby podporoval synchronizaci s požadovanou mírou zabezpečení a kontroly, musíme použít vlastní plánovač.

6 Architektura řešení

Implementace uzlu SAN je provedena v jazyce Java. Jednotlivé komponenty jsou od sebe navzájem odstíněné pomocí rozhraní. Třída `san.core.Process` slouží jako reprezentace jednotlivých procesů (uživatelských aplikací a kapsulí). Tato třída je použita jako zprostředkovatel komunikace mezi jednotlivými komponentami. Umožňuje v kontextu daného procesu komunikovat a proces si navzájem mezi komponentami předávat.

Naším úkolem je vytvořit nový plánovač a upravit interpretr tak, aby dokázal vylepšené plánování podporovat.

6.1 Prototyp architektury

Na počátku návrhu architektury je vhodné si vyzkoušet zamýšlené řešení plánovače a fiber vláken pomocí jednoduchého prototypu.

6.1.1 Jednoduchý plánovač vláken Java

Jednoduchou implementaci plánovače v jazyce Java lze zajistit pomocí priority vláken. Aplikace jsou spouštěny v nových vláknech jazyka Java. Plánovač je také spuštěn ve vlastním vlákně, které má nastavenu nejvyšší možnou prioritu. Při naplánování vlákna plánovač zvyšuje jeho prioritu, při odplánování ji naopak snižuje na minimální prioritu.

Ačkoliv samotná vlákna plánuje standardní plánovač JRE, náš jednoduchý plánovač nastavením priorit určí, které vlákno má standardní plánovač upřednostnit. Tento koncept je velmi podobný konceptu plánovače z projetu Smart Packets.

Na následujícím fragmentu kódu je ukázka implementace jednoduchého plánovače.

```
class SimpleScheduler extends Thread {

    void run() {
        this.setPriority(Thread.MAX_PRIORITY);
        while (…) {
            Thread selectedThread = selectThreadToSchedule();
            selectedThread.setPriority(Thread.NORM_PRIORITY);
            Thread.sleep(TIME_QUANTUM);
            selectedThread.setPriority(Thread.MIN_PRIORITY);
        }
    }
}
```

Metoda `selectThreadToSchedule()` se stará o výběr vlákna, které má získat na následující časové kvantum procesor. Zde se bude nacházet implementace plánovacího algoritmu.

Tato metoda plánování bohužel neposkytuje našemu plánovači úplnou kontrolu nad vykonáváním vláken. Nelze zaručit, že námi nenaplánovaná vlákna plánovač JRE nespustí. Pokud například se naplánované vlákno za běhu zablokuje, plánovač JRE může naplánovat

a spustit jiné vlákno dříve, než náš plánovač provede své plánování. Dále toto řešení nelze použít u víceprocesorových konfigurací, plánovač JRE může spustit námi nenaplánovaná vlákna na jiných procesorech. Není tedy zaručeno vzájemné vyloučení a náš plánovač nemůže poskytovat vlastní synchronizační prostředky.

6.1.2 Jednoduchá implementace vláken Java

Pro získání plné kontroly nad vykonáváním vláken je potřeba se při jejich plánování nespoléhat na standardní vlákna JRE. Vytvoření vlastní jednoduché implementace fiber vláken pro plánovač z předchozí kapitoly umožňuje plně řídit běh jednotlivých vláken.

Na následujícím fragmentu kódu je jednoduchá ukázka implementace fiber vláken a jejich plánování.

```
interface IFiberThread {

    void executeOneInstruction();
}

class SimpleFiberScheduler extends Thread {

    void run() {
        while (…) {
            IFiberThread selectedThread = selectThreadToSchedule();
            long startTime = getCurrentTime();
            while (getCurrentTime() < startTime + TIME_QUANTUM) {
                selectedThread.executeOneInstruction();
            }
        }
    }
}
```

Toto řešení již umožňuje plnou kontrolu nad běžícími vlákny a plánovač již může poskytovat vlastní synchronizaci. Stále však toto řešení nepodporuje používání více souběžně běžících interpretrů.

6.1.3 Podpory více souběžně běžících interpretrů

Pro zajištění více souběžně běžících interpretrů je potřeba přenést vykonávání instrukcí z vlákna plánovače do jednotlivých vláken interpretrů. Plánovač bude jen v určitém časovém intervalu přiřazovat interpretrům vlákna, kterým má být přidělen procesor. Interpretr bude následně vykonávat instrukce přiděleného vlákna, dokud mu plánovač nepřiřadí nové vlákno.

Na následujícím fragmentu kódu je ukázka plánování a souběžného vykonávání více fiber vláken.

```
interface IInterpreter {

    void runThread(FiberThread thread);
}

class SimpleInterpreter extends Thread implements IInterpreter {

    IFiberThread scheduledThread;

    void runThread(IFiberThread thread) {
        scheduledThread = thread;
    }

    void run() {
        while (…) {
            IFiberThread thread = scheduledThread;
            thread.executeOneInstruction();
        }
    }
}

class MultiFiberScheduler extends Thread {

    public void run() {
        while (…) {
            for (int i = 0; i < CPU_COUNT; i++) {
                IFiberThread selectedThread = selectThreadToSchedule();
                interpreter[i].runThread(selectedThread);
            }
            Thread.sleep(TIME_QUANTUM);
        }
    }
}
```

Toto řešení umožňuje plánování a souběžné vykonávání více fiber vláken.

6.1.4 Závěr plynoucí z prototypu

Díky použití prototypu bylo ověřeno, že zvolená architektura plánovače a fiber vláken poskytuje požadovanou funkcionalitu. Zvolenou architekturu můžeme použít při implementaci plánovače a fiber vláken pro uzel SAN.

6.2 Shrnutí cílů implementace

Je potřeba vytvořit nový plánovač splňující rozhraní `san.core.IScheduler` a nahradit jím předchozí plánovač. Dále je třeba upravit SAN interpretr tak, aby byl schopen na požádání pozastavit vykonávání procesu a uložit jeho stav pro pozdější opětovné spuštění (implementace fiber vláken). Rozhraní `san.interpreter.IInterpreter` bude použito pro ovládání interpretru. Dále je upravit interpretr pro běh ve více instancích a umožnit tak paralelní vykonávání více procesů současně.

Vzhledem k tomu, že funkčnost interpretru je nezávislá na ostatních komponentách, je výhodné začít s implementací právě u interpretru a až následně sestrojit plánovač, který bude využívat jeho rozšířených služeb.

6.3 Použití kolekcí platformy Java

Ve většině případů bude pro uchovávání procesů použita kolekce spojového seznamu (`java.util.LinkedList` přístupný přes rozhraní `java.util.Queue`). Implementace spojového seznamu je výhodná pro použití jako FIFO fronta, protože operace vkládání na konec fronty a vyjímání z čela fronty má konstantní výpočetní složitost O(1). Další výhoda spojového seznamu je v rychlém vyjímání libovolných prvků, opět s konstantní výpočetní složitostí O(1). Další častou činností nad seznamem procesů je jejich postupné procházení, které je pro všechny kolekce vždy O(n).

Ve speciálních případech, kde je potřeba použít také vyhledávání, je použita kolekce tabulky (`java.util.HashMap` přístupná přes rozhraní `java.util.Map`).

6.4 Synchronizace

V původním konceptu plánovače byl použit jeden monitor, který chránil celou metodu zajišťující plánování. Použití jednoho monitoru pro jednu velkou kritickou sekci je nevýhodné, lepší řešení je kritickou sekci rozdělit a použít několik monitorů. V případě velké kritické sekce a jednoho monitoru je potřeba také zahrnout do kritické sekce některé části kódu, které nevyžadují synchronizaci. Použití více monitorů umožňuje přesněji vymezit kritickou sekci a synchronizovat jen potřebné části kódu. To zajistí zvýšení výkonnosti, neboť vlákna budou schopna vykonávat vetší část kódu souběžně.

Proměnné, ke kterým je přistupováno z více různých vláken, jsou deklarovány s klíčovým slovem `transient`. Klíčové slovo `transient` zajistí, že hodnota proměnné není ukládána ve vyrovnávací paměti vláken, ale je vždy uložena do hlavní paměti přímo.

6.5 Interpretr

Původní interpretr vykonává spuštěný proces v jednom novém vlákně. Volání metod interpretované aplikace je implementováno pomocí rekurze, ve které interpretr volá metodu `InterpreterBytecode.Run(…)` pro započetí interpretace metody aplikace.

6.5.1 Zásobník volání

Naším úkolem zde je nahradit rekurzivní volání metody `Run(…)` v původním interpretru `InterpreterBytecode` vlastní implementací zásobníku volání metod. Tento zásobník bude ukládán v třídě `FiberInterpretingContext`, která je interpretru přístupná pomocí rozhraní `san.interpreter.IInterpretingContext`. Reference na toto rozhraní je uložena ve třídě `san.core.Process` odpovídající danému procesu. Každý proces má tedy svůj vlastní kontext interpretace, ve kterém je uložen zásobník volání.

V původním návrhu uzlu SAN se s využitím kontextu interpretace již počítalo, ale nebyl nikdy implementován a rozhraní `san.interpreter.IInterpretingContext` bylo doposud nevyužité.

Do zásobníku volání se vkládá rámec, který obsahuje následující položky:

- Bytecode vykonávané metody.
- Reference na instanci třídy `HardClass`, které volaná metoda patří.
- Interpretované reference.
- Datový zásobník metody.
- Lokální proměnné.
- Ukazatel pro zápis návratové hodnoty do datového zásobníku.
- Instrukční ukazatel (program counter) pro právě vykonávanou metodu.

Kromě zásobníku volání je v kontextu volání uložena také reference na `ClassManager` příslušný danému procesu.

6.5.2 Nahrazení volání metod

Nyní je nutné zajistit, že veškeré volání metody pro spuštění interpretace `InterpreterBytecode.Run(…)` bude nahrazeno vytvořením nového rámce v zásobníku volání metod.

Toto volání se objevuje jen ve dvou třídách. Ve třídě `HardClass` v metodě `RunMethod(…)`, která slouží k interpretaci volání metod nebo konstruktorů. Dále se objevuje ve třídě `ClassManager` v metodě `loadClass(…)`, která slouží k načtení nové doposud nepoužité třídy (je volána metoda `cinit()` pro inicializaci třídy a jejích statických atributů), a v metodě `runMethod(…)` pro spuštění metody `main(…)` uživatelských aplikací a kapsulí.

V případě třídy `HardClass` je volání metody `RunMethod(…)` vždy podmíněno instrukcí `INVOKEINTERFACE`, `INVOKESPECIAL`, `INVOKESTATIC` nebo `INVOKEVIRTUAL`. V Případě třídy `ClassManager` je volání metody `RunMethod(…)` provedeno kdykoliv, kdy je načtena doposud neinicializovaná třída.

Třída `ClassManager` je volána v mnoha různých částech kódu při vykonávání různých instrukcí, obvykle navíc opakovaně a rekurzivně. Toto způsobuje komplikace při určování pořadí vykonávání metod, které byly do zásobníku volání vloženy.

6.5.3 Spekulativní vkládání rámců zásobníku

Pokud voláme metodu, která patří třídě, která ještě nebyla inicializována (metoda `cinit()`) nebo nebyla ještě vytvořena její instance (volání konstruktoru třídy), je nutné zaručit správnou posloupnost vykonávání jednotlivých metod.

Rámec je nutné do zásobníku volání uložit vždy okamžitě po instrukci pro volání metody. Následně může dojít k dalšímu volání metody `runMethod(…)` třídou `ClassManager`, a to i opakovaně nebo rekurzivně, například při získávání bytecode dané metody, který může vyžadovat inicializaci třídy nebo více tříd.

Po vložení rámce do zásobníku volání se nevzdáme jeho reference, ale naopak ji využijeme v závěru původního volání metody `runMethod(…)`, kdy tento rámec naplníme odpovídajícími atributy (například získaným bytecode metody).

Pokud na závěr nedojde k získání bytecode metody, například pokud je metoda nativní nebo se jedná o referenční metodu, je metoda spuštěna pomocí reflexe a rámec zásobníku není nenaplněn. Později, když na tento rámec přijde v zásobníku řada, je rámec zahozen a místo něj je použit další neprázdný rámec. Tímto je zajištěno správné pořadí interpretování jednotlivých volání metod.

Pro nativní metody neexistuje bytecode, jsou implementovány nativně pro každou platformu JRE zvlášť mimo jazyk Java (například v assembleru). Referenční metody jsou součástí tříd (`SoftReference`, `SoftClass`), jejichž instanci získává interpret již vytvořenou uzlem SAN. Začlenění těchto metod do interpretace je zajištěno pomocí nástroje reflexe, který umožňuje jejich volání. Takto například interpretovaná aplikace přistupuje k aplikačnímu rozhraní SAN.

6.5.4 Nahrazení interpretru

Dalším krokem je vytvoření třídy `FiberInterpreterBytecode`. Tato třída obsahuje metodu `oneStep(…)`, která zajistí provedení jedné instrukce z bytecode.

Metoda `oneStep(…)` vezme rámec z vrcholku zásobníku volání, nastaví všechny požadované atributy a vykoná jednu instrukci bytecode. Pokud instrukce byla poslední a metoda skončila, zajistí uložení návratové hodnoty na správné místo datového zásobníku. Pokud skončila hlavní metoda `main()`, je návratová hodnota uložena do třídy `san.core.Process`.

Pro urychlení vykonávání metody `oneStep(…)` nedochází k manipulaci se zásobníkem volání při každé instrukci, ale jen je na počátku volání otestováno, zda se zásobník nebo interpretovaný proces od poslední instrukce nezměnil. Pokud ano, je rámec ze zásobníku vyjmut a všechny požadované atributy jsou znovu nastaveny. Touto drobnou optimalizací je zajištěno, že pro vykonání jedné instrukce je zapotřebí minimum režie, změny jsou provedeny jen při změně vykonávané metody (volání nové metody) nebo procesu (došlo k přeplánování).

Tímto byla odstraněna rekurze z implementace volání metod. Původní implementace `InterpreterBytecode` již není po změně přístupu k volání metod použitelná.

6.5.5 Statické atributy interpretovaných tříd

V případě přístupu ke statickým atributům třídy, která ještě nebyla třídou `ClassManager` načtena, docházelo k problému. Nejdříve byla volána instrukce pro přístup ke statickému atributu (`PUTSTATIC` a `GETSTATIC`). Následně byla načtena třída a provedena instrukce vůči dané statické proměnné, a až v dalším kroku bylo zahájeno vykonávání metody `cinit()`, která statické proměnné inicializovala. Toto je způsobeno přechodem od rekurze k zásobníku volání metod, který nemůže v tomto případě dodržet požadované pořadí operací.

Řešením bylo vytvořit speciální rámec zásobníku volání, který umožňuje provést instrukci se statickými proměnnými až po vykonání metody `cinit()` pro inicializaci těchto statických proměnných. Tento rámec je vytvářen ve třídě `ClassManager` při volání metod `getStaticVariable(…)` a `setStaticVariable(…)`.

Podobným způsobem bylo nutné ošetřit inicializaci třídy `java.lang.System`, pro kterou byl také vytvořen zvláštní rámec volání metod. Tento rámec je využit v třídě `ClassManager` v metodě `Init()`.

Uvedené speciální rámce pro přístup ke statickým proměnným jsou třídou `FiberInterpreterBytecode` po vyjmutí ze zásobníku rozeznány a odpovídajícím způsobem vykonány.

6.5.6 Implementace rozhraní interpretru

Aby bylo možné využít nové funkčnosti interpretru, je potřeba nahradit stávající implementaci rozhraní `san.interpreter.IInterpreter`, která interpretr řídí. Původní implementace `SanInterpreter` spouštěla každý proces ve vlastním vlákně a neumožňovala pozastavení a změnu interpretovaného procesu.

Z daného rozhraní využívala tato třída jen metodu `runProcess(…)` pro počáteční spuštění interpretace, opakované volání metody `runProcess(…)` a ostatních metod `terminate()`, `getProecess()` a `isInterpreting()` nebyly využity. Proto bude tato třída nahrazena novou implementací, třídou `SanFiberInterpreter`, která plně využije dané rozhraní a nabídne plánovači skrze něj svoji rozšířenou funkčnost.

Třída `SanFiberInterpreter` bude sama spuštěna v novém vlákně, a za pomoci sdílené proměnné bude přijímat referenci na proces, který má interpretovat. K manipulaci s touto sdílenou proměnou budou sloužit výše uvedené metody z rozhraní interpretru. Sdílenou proměnnou mohou nastavovat jen metody rozhraní interpretru volané plánovačem. Sám interpretr tuto proměnnou nijak nemodifikuje, jen ji čte a následně interpretuje daný proces.

Předchozí implementace plánovače a interpretru předpokládala, že když je interpretace procesu skončena, může sám interpretr tuto proměnnou modifikovat. Tato verze řešení však nebyla z původní implementace interpretru přejata, neboť vyžadovala vzájemnou synchronizaci plánovače a interpretru, která by mohla mít vliv na výkon. Nová (v předchozím odstavci popsaná) verze předpokládá, že interpretr sám tuto proměnnou nemodifikuje, a proto není z jeho strany potřeba žádná synchronizace. Pokud je v průběhu vykonávání jedné instrukce tato poměná změněna, interpretr se to dozví až po vykonání aktuální instrukce a na změnu zareaguje s určitým zpožděním, ale bez nutnosti vzájemného čekání. Plánovač nemusí čekat na potvrzení převzetí naplánovaného procesu.

Aby bylo možné zajistit souhru více různých instancí interprrtru, je každý proces doplněn o vlajku (flag), která určuje, zda je proces právě vykonáván v nějakém interpretru. Dokud je vlajka nastavena, nesmí jiný interpretr začít proces vykonávat. Tímto je zajištěno vzájemné vyloučení nad kontextem interpretace.

Řízení interpretace je možné popsat následujícím souborem pravidel:

1) Ze sdílené proměnné získej aktuálně naplánovaný proces.
2) Porovnej naplánovaný a doposud vykonávaný proces, pokud se liší, tak:
    a) Doposud vykonávaný proces nastav do stavu připravený.
    b) Uvolni vlajku interpretace doposud vykonávaného procesu.
    c) Čekej, dokud má naplánovaný proces nastavenou vlajku interpretace.
    d) Pokud není naplánovaný proces ve stavu připravený, tak informuj plánovač a uspi se, po probuzení pokračuj bodem 1).
    e) Označ naplánovaný proces vlajkou interpretace.
    f) Naplánovaný proces nastav do stavu běžící.
    g) Do proměnné držící aktuálně vykonávaný proces ulož naplánovaný proces.
3) Pokud není naplánován žádný proces nebo proces není ve stavu běžící, tak informuj plánovač a uspi se, po probuzení pokračuj bodem 1).
4) Pokud se jedná o první spuštění procesu, tak proveď inicializaci kontextu interpretace.
5) Vykonej jednu instrukci.
6) Pokud byla vykonána poslední instrukce metody a zároveň je zásobník volání prázdný, tak proces nastav do stavu dokončený.
7) Pokud proces není ve stavu běžící, informuj plánovač.
8) Pokud proces skončil chybou, tak proces nastav do stavu zastavený a informuj plánovač.
9) Vrať se na bod 1).

Probuzení procesu je zajištěno libovolným voláním jeho rozhraní z plánovače. V bodech a) až g) není zdůrazněno otestování, zda je nějaký proces skutečně naplánován.

Pokud není zadaný naplánovaný nebo doposud vykonávaný proces (obsahuje hodnotu null), tak se tyto kroky přeskakují.

Pokud bychom vynechali režii na změnu procesu vůči plánování a soustředili bychom se jen na samotné vykonávání procesu, předchozí soubor pravidel lze přepsat následovně:

1) Zkontroluj, zda nebyl plánovačem naplánován nový proces.
2) Pokud není naplánován žádný proces, tak se uspi, po probuzení pokračuj bodem 1).
3) Vykonej jednu instrukci procesu.
4) Vrať se na bod 1).

Tyto čtyři body jsou velmi důležité na návrh a implementaci, neboť se jedná o úzké hrdlo interpretru. Do tohoto úzkého hrdla je nutné také uvažovat režii na manipulaci se zásobníkem volání metod. Pokud by byly implementovány neefektivně, docházelo by ke znatelnému zpomalení výkonu interpretru.

6.5.7 Více instancí interpretru

Interpretr byl od začátku koncipován tak, aby mohl souběžně běžet ve více instancích. Každá instance třídy `SanFiberInterpreter` zapouzdřuje své vlastní vlákno provádějící smyčku interpretace, ke kterému je poskytnut přístup pomocí rozhraní interpretru, které tato třída implementuje.

O přístup k funkcím jedné instance interpretru se stará rozhraní interpretru, dále je však nutné provézt změny ve správci interpretrů, aby mohl poskytnout plánovači přístup k jednotlivým instancím. Postačuje pouze rozšířit správce interpretrů o metodu `getSanFiberInterpreter(int cpuIndex)`, která zajistí vrácení konkrétní požadované instance. Jednotlivé interpretry jsou identifikovány celým číslem indexovaným od nuly.

Vlákna interpreterů jsou zavedena a spuštěna při startu uzlu SAN. Během provozu interpretr nevytváří žádná nová vlákna.

Počet spuštěných instancí interpretru lze nastavit v konfiguračním souboru uzlu SAN.

6.5.8 Interpretce sdílení statických proměnných

Původní implementace interpretru ignorovala sdílení statických proměnných. Pro zajištění meziprocesové komunikace je však sdílených proměnných potřeba. Proto bylo nutné sdílení statických proměnných do interpretru doimplementovat.

Sdílet nelze interní objekty interpretru `HardClass` a `HardReference`. Tyto objekty si vytváří každý proces zvlášť za pomoci své vlastní instance třídy `ClassManager` a proto je nelze mezi jednotlivými procesy sdílet bez rizika ovlivnění kontextu vykonávání cizího procesu. Pokud jsou tyto objekty přesto uloženy do statických proměnných, pokusí se

interpretr o jejich převod na objekty `SoftClass` a `SoftReference`. Pokud je převod úspěšný, lze tyto objekty již bezpečně sdílet pomocí statických proměnných. V současné verzi interpretru tento převod nefunguje pro pole. Objekt pole nelze převést na měkkou referenci, a proto jej nelze sdílet.

Sdílení primitivních datových typů je zajištěno hodnotou (nikoliv odkazem), proto není v tomto případě potřeba provádět jakékoliv převody.

Samotné sdílení zajišťuje třída `StaticManager`, která překrývá volání metod `getStaticVariable()` a `setStaticVariable()` třídy `HardClass`. Třída `StaticManager` existuje jako singleton, čímž je zajištěno sdílení jejího obsahu pro všechny instance interpretrů.

Každá statická proměnná náleží své třídě, která je jednoznačně identifikována jménem a balíkem. Hodnota statické proměnné je uložena pod klíčem, který se skládá z balíku a jména třídy a jména proměnné.

Sdíleny jsou jen statické proměnné uživatelských aplikací a kapsulí. Statické proměnné tříd standardní knihovny JRE a uzlu SAN nejsou do tohoto sdílení z důvodu bezpečnosti zahrnuty. Toto opatření se týká následujících balíků:

- `java.*`
- `javax.*`
- `org.*`
- `sun.*`
- `san.*`

6.5.9 Přetížení synchronizace

Synchronizace v jazyce Java je řešena jen pomocí dvou instrukcí – `MONITORENTER` a `MONITOREXIT`. V původní implementaci interpretru nebyly tyto instrukce podporovány. Nová implementace třídou `FiberInterpreterBytecode` již tyto instrukce zohledňuje a jejich volání předává plánovači, který nyní bude zajišťovat synchronizaci procesů.

Díky tomuto lze použít vlastní implementaci synchronizace pomocí standardního zápisu kritické sekce `synchronized (…) {…}`.

6.5.10 Přetížení volání metod

Předchozí interpretr mohl využívat synchronizační prostředky jazyka Java a volat blokující metody. Interpretace synchronizace je prováděna pomocí reflexe.

Protože každý proces běžel v novém vlákně a měl vlastní instanci interpretru, bylo možno tato vlákna pozastavovat a blokovat. Nyní však každý proces neběží v novém vlákně, ale procesy jsou vykonávány jednou instancí interpretru, o kterou se střídají. Pokud by

interpretr zavolal například metodu `Thread.sleep(…)`, došlo by k pozastavení celého interpretru. Pozastavený interpretr by přestal reagovat a nemohl by vykonávat jiné procesy.

Proto je nutné přetížit volání těchto metod a jejich volání přesměrovat na náš plánovač. O toto přesměrování se stará třída `MethodCallOverride`. Tato třída na počátku volání metody `runMethod()` třídy `HardClass` podle jména volané metody rozhodne, zda bude volání metody přetíženo. Jednotlivá přetížení jsou určena pomocí potomků třídy `IMethodCall`.

Přetíženy jsou následující metody:

- `java.lang.Thread.sleep(…)` – Uspání procesu na určitý čas.
- `java.lang.Thread.yield()` – Odebrání zbytku přiděleného časového kvanta.

Dále je přetíženo volání metod pro práci s podmínkovými proměnnými jazyka Java. Jejich zavolání by způsobilo také zablokování interpretru. Pro zajištění synchronizace bude potřeba implementovat vlastní mechanizmus podmínkových proměnných. Metody `wait(…)`, `notify()`a `notifyAll()` třídy `java.lang.Object` nelze v současné implementaci interpretru použít. Jejich volání není nyní povoleno a způsobuje při interpretaci výjimku.

6.5.11 Monitory

Pro použití bloku synchronizace je potřeba určit objekt, nad kterým bude synchronizace provedena.

Stejně jako v případě sdílení pomocí statických proměnných, zde nelze použít objekty `HardClass` a `HardReference`. Jednoduché řešení tohoto problému je používat jako monitory jen objekty uložené ve statických proměnných, které jsou vždy již převedené na měkkou referenci.

Toto řešení není nijak omezující. Pokud používáme monitor, potřebujeme jej vždy sdílet mezi více různými procesy. V opačném případě není potřeba nad ním synchronizovat, protože se nejedná o sdílený objekt.

6.5.12 Speciální procesy

Pro doručení dat z příchozí kapsule do aplikace je potřeba interpretovat kód metody `receiveData(…)` z poskytnutého rozhraní `ReceiveDataListener`, které aplikace musí implementovat. V původní verzi byla tato metoda spuštěna v novém vlákně, ve kterém si vytvořila novou instanci interpretru a provedla doručení dat z kapsule.

Tento postup nyní není možný, metodu nelze okamžitě spustit. Je potřeba volání této metody obalit novým procesem a tento proces naplánovat a přidělit interpretru. Volání a spuštění těchto procesů je zajištěno pomocí potomků třídy `ISpecialCall`. Doručení dat z kapsule do aplikace zajišťuje oddědděná třída `ReceiveDataCall`.

6.6 Plánovač

Pro využití nové funkčnosti interpretru je potřeba vylepšit plánovač. Na rozdíl od interpretru, k jehož provozu je potřeba 25 různých tříd, je rozsah problematiky plánovače skromnější. Původní plánovač se skládá jen z jedné třídy implementující rozhraní plánovače a zapouzdřující tři jednoduchá vlákna. Proto si můžeme dovolit vytvořit novou implementaci plánovače zcela od začátku.

Původní třída plánovače `san.core.Scheduler` proto bude nahrazena její novou implementací, třídou `san.core.fiber.scheduler.FiberScheduler`. Protože obě třídy implementují rozhraní plánovače `san.core.IScheduler`, přes které komunikují s ostatními komponentami, je možné je zaměnit bez nutnosti jakýchkoli změn.

6.6.1 Činnost plánovače

Plánovač se v původním konceptu skládal ze tří vláken:

- Vlákno pro přípravu kapsulí.
- Vlákno pro plánování kapsulí.
- Vlákno pro plánování aplikací.

Pro komunikaci mezi těmito vlákny a ostatními komponentami uzlu SAN slouží rozhraní plánovače a jeho metody. Následuje výčet metod z rozhraní plánovače, které jsou z hlediska plánování pro nás důležité:

- `runApplication(…)` – Metoda pro spuštění nové aplikace. Plánovač pomocí této metody obdrží nový proces, který má být spuštěn.
- `waitForApplication(…)` – Metoda pro vyčkání na konec aplikace. Aplikace, která tuto metodu zavolá, je uspána, dokud zvolený proces neskončí.
- `intepretingEnded(…)` – Metoda pro oznámení konce interpretace. Interpretr touto metodou informuje plánovači, že interpretace daného procesu již skončila.

Protože původní plánovač pouze spouštěl příchozí kapsule a aplikace v nových vláknech, bylo možné jejich plánování provádět separátně. Stejně tak kolekce pro uchovávání kapsulí byly oddělené od kolekcí pro uchovávání uživatelských aplikací. Nový plánovač bude při svém plánování vyžadovat jednotný přístup ke kapsulím a aplikacím. Kapsule a uživatelské aplikace jsou z hlediska plánovače procesy, které se liší jen způsobem, jakým jsou plánovači předány. Proto je možné plánování kapsulí a aplikací spojit.

Plánování kapsulí a aplikací budeme provádět v jednom vlákně a ukládat je budeme do společných kolekcí.

Vlákno pro přípravu kapsulí zajišťuje hlídání počtu běžících kapsulí, posílání požadavků na chybějící kód kapsulí a čekání na doručení vyžádaného kódu. Funkčnost tohoto

vlákna nijak nesouvisí s plánováním, a proto je jeho původní implementace převzata a jen upravena pro nový způsob ukládání a spouštění kapsulí.

6.6.2 Nové stavy procesu

Aby bylo možné zajistit nové funkce poskytované plánovačem, bylo potřeba rozšířit množinu stavů procesu o stavy popsané v kapitole 4.1. Proces se může nacházet v jednom z následujících stavů:

- `READY_TO_RUN` – Proces je připravený ke spuštění a čeká na přidělení procesoru.
- `RUNNING` – Proces je spuštěný, byl mu přidělen procesor a právě probíhá výpočet.
- `FINISHED` – Proces dokončil svůj výpočet a skončil.
- `STOPPED` – Výpočet procesu byl přerušen neošetřenou výjimkou.
- `ASLEEP` – Proces je uspán na určitý čas.
- `BLOCKED_CONDITION` – Proces je blokován na podmínkové proměnné.
- `BLOCKED_MONITOR` – Proces je blokován čekáním na monitoru.
- `BLOCKED_JOIN` – Proces je blokován čekáním na skončení jiného procesu.

Stav blokovaný je realizován jen jedním stavem. Důvodem je původní koncepce, která nepředpokládala takovéto využití tohoto stavu a kterou je potřeba dodržet. Důvod blokace procesu je uložen ve zvláštním výčtu `BlockedReason`.

6.6.3 Komunikace plánovače a interpretru

Na původním konceptu nebyla provedena žádná změna. Plánovač vlastní správce interpretrů, který vlastní jednotlivé interpretry. Pokud interpretr dokončí interpretaci svého procesu, oznámí to pomocí správce interpretrů plánovači metodou `interpretingEnded(…)` z rozhraní plánovače.

V původním konceptu metoda `interpretingEnded(…)` přímo přistupovala ke kolekcím, ve kterých byly procesy uloženy. Pokud například proces dokončil výpočet, volání této metody přistoupilo k seznamu běžících procesů a daný proces z ní odstranilo.

Tento koncept vyžadoval důsledné vzájemné vyloučení, kterému se pokusíme vyhnout. Místo sdílení kolekcí pro správu procesů je použito zasílání zpráv. Plánovací vlákno vždy na začátku iterace zkontroluje, zda nebyl vznesen nový požadavek. Pokud ano, provede potřebné akce nad kolekcemi s uloženými procesy. Díky tomuto postupu je ke kolekcím s procesy přistupováno jen z plánovacího vlákna, a proto není potřeba je synchronizovat.

Předávání zpráv je řešeno pomocí třídy `FiberSchedulerMailbox`, která v sobě zapouzdřuje schránky. Schránky jsou určeny pro nové procesy určené ke spuštění a procesy, které se vzdaly procesoru nebo byly zablokovány na synchronizaci. Schránky také zajišťují upozornění na nové zprávy. Po příchodu nové zprávy je probuzeno plánovací vlákno.

6.6.4 Volání synchronizace

Implementace metod pro volání synchronizačních prostředků (monitory, podmínkové proměnné a uspání vlákna) jsou uloženy ve třídě `san.core.Process`. Pokud je proces zavolá, metoda uloží do kontextu procesu informaci o daném volání. Interpretr následně oznámí plánovači konec vykonávání procesu voláním metody `interpretingEnded(…)`, která zavolá modul pro synchronizační monitory nebo podmínkové proměnné. Modul provede obsloužení původního volání a rozhodně, zda proces smí pokračovat ve vykonávání, nebo zda byl zablokován. V případě zablokování je probuzen plánovač, který provede přeplánování interpretru. Pokud proces zablokován nebyl (například zabrání volného monitoru), není plánovač probuzen a interpretr pokračuje ve vykonávání procesu.

Volání metody `interpretingEnded(…)` neznamená konec procesu. Znamená nemožnost vykonávat proces dále, dokud plánovač nebo synchronizační moduly nerozhodnou, zda proces byl nebo nebyl zablokován či uspán.

Pokud by plánovač nebyl po ukončení procesu interpretrem probuzen, interpretr by byl po zbytek časového kvanta nevyužitý. Naším cílem je ale maximálně využívat procesor, proto je nutné plánovač probudit a přidělit interpretr jinému procesu. Stejně tak odezva plánovače je urychlena, pokud plánovač reaguje okamžitě, nikoliv až po vypršení celého časového kvanta.

6.6.5 Architektura plánovače

Plánovací vlákno je periodicky spouštěno. Interval spouštění je roven nastavenému časovému kvantu. Pokud plánovač obdrží požadavek na spuštění nového procesu nebo interpretr oznámí konec vykonávání procesu, je okamžitě spuštěno nové plánování.

Plánovač se skládá z následujících částí:

- Plánovací strategie – Vybírá proces, kterému bude přidělen procesor. Výběr je realizován zvoleným plánovacím algoritmem.
- Optimalizační mezivrstva pro více procesorů – Zajišťuje optimální rozložení procesů na procesory.
- Zavaděč (dispatcher) – Spouští a pozastavuje procesy podle rozhodnutí plánovací strategie.
- Modul pro správu spánku – Zajišťuje uspání procesů na danou dobu. Dokud nevyprší doba spánku, modul nedovolí naplánování procesu.
- Modul pro správu podmínkových proměnných – Zajišťuje synchronizaci pro podmínkové proměnné. Dokud proces čekající na podmínkové proměnné neobdrží signál k opuštění podmínky, modul nedovolí jeho naplánování.
- Modul pro správu synchronizačních monitorů – Zajišťuje synchronizaci pro monitory. Pokud je proces blokovaný monitorem, modul nedovolí jeho naplánování.

- Modul pro detekci uvíznutí – Periodicky kontroluje, zda nedošlo k uvíznutí. Pokud ano, řeší uvíznutí násilným ukončení vybraných procesů.
- Modul pro prevenci uvíznutí – Pokud procesy dopředu oznámí, jaké monitory budou zabírat, zajistí tento modul prevenci uvíznutí. Pomocí bankéřova algoritmu modul kontroluje, zda jednotlivá zabírání monitorů nevedou k uvíznutí. Pokud ano, nedovolí modul zabrání dalších monitorů a proces je pozastaven, dokud se situace nezmění.

Hlavní část plánovače je implementovaná podle kapitoly 4.2 v metodě `schedule()` uvnitř třídy `FiberScheduler`. Metoda je periodicky spouštěna plánovacím vláknem. Uspání vlákna po dobu časového kvanta je zajištěno podmínkovou proměnnou. Vlákno čeká nad podmínkou voláním metody `Object.wait(long timeout)`. Časový limit čekání je časové kvantum. Pokud je třeba plánovač předčasně probudit, je volána metoda `Object.notify()`. Předčasné probouzení plánovacího vlákna zajišťuje třída `FiberSchedulerMailbox`, která zapouzdřuje podmínkovou proměnnou.

6.6.6 Vnitřní struktury plánovače

Plánované procesy je potřeba ukládat a organizovat. Plánovač procesy organizuje do následujících struktur:

- Seznam všech procesů – V tomto seznamu jsou uloženy všechny plánovatelné procesy, tedy procesy připravené, běžící, spící, čekající a blokované.
- Seznam běžících procesů – V tomto seznamu jsou uloženy jen právě vykonávané procesy. Seznam je podmnožinou seznamu všech procesů a jeho velikost je vždy menší nebo rovna počtu dostupných interpretrů.
- Seznam běžících Bean Shell procesů – Protože SAN interpretr nepodporuje Swing rozhraní (grafická konzole uzlu SAN), je potřeba použít Bean Shell interpretr a začlenit jej do plánování. Procesy Bean Shell nelze plánovat jako fiber procesy, proto je nutné je ukládat do vlastních kolekcí a odpovídajícím způsobem s nimi pracovat.

Tyto tři seznamy jsou přístupné jen z vlákna plánovače, ostatní komponenty uzlu SAN komunikují s plánovačem pomocí zasílání zpráv. Jako schránka na zaslané zprávy je použita třída `FiberSchedulerMailbox`.

Ačkoliv by se mohlo zdát, že by bylo výhodnější procesy ukládat do seznamů podle jejich aktuálního stavu (připravené, běžící, spící, čekající a blokované), zvolená implementace je výhodnější. Při každém přeplánovaní je potřeba přistoupit ke každému procesu. Běžícím procesům je potřeba přičíst spotřebu naposledy spotřebovaného časového kvanta, připraveným procesům je naopak naposledy spotřebované časové kvantum zkráceno, u spících procesů je potřeba zkontrolovat, zda již nevypršel časový limit jejich spánku.

Přesouvání procesů mezi kolekcemi v závislosti na změnách jejich stavů by vyžadovalo režii navíc pro manipulaci s kolekcemi.

6.6.7 Algoritmus plánovače

Algoritmus plánovače lze popsat následujícím souborem pravidel:

1) Uspi se na dobu časového kvanta. Pokud bude během spánku zachyceno přerušení, ihned se probuď.
2) Pokud nebyla zaslána žádná nová zpráva (nový proces připravený ke spuštění nebo ukončení vykonávaného procesu) a seznam všech procesů je prázdný, vrať se na bod 1).
3) Aktualizuj běžícím procesům spotřebu časového kvanta.
4) Pokud byly zaslány nové procesy, nastav je do stavu připravený a vlož je do seznamu všech procesů.
5) Procházej seznam všech procesů:
    a) Pokud je proces ve stavu dokončený, odstraň ho ze seznamu.
    b) Pokud je proces ve stavu připravený, zkrať jeho spotřebované časové kvantum.
6) Pomocí plánovací strategie vyber proces, kterému bude přidělen procesor. Vyber nejvýše tolik procesů, kolik je spuštěno interpretrů.
7) Proveď optimalizaci přiřazení naplánovaných procesů na jednotlivé interpretry.
8) Procházej seznam dvojic (interpretr, naplánovaný proces):
    a) Pokud je právě interpretovaný proces zároveň naplánovaným procesem, tak nedělej nic a přejdi na další dvojici.
    b) Pokud v interpretru běží jiný proces, tak jej pozastav, nastav jeho stav na připravený a odeber jej ze seznamu běžících procesů.
    c) Naplánovaný proces nastav do stavu běžící, vlož jej do seznamu běžících procesů a spusť jej.
9) Vrať se na bod 1).

6.6.8 Plánovací strategie

Plánovací strategie má za úkol vybrat proces, kterému bude na následující časové kvantum přidělen procesor. Abstraktní třída `ISchedulerStrategy` slouží jako rozhraní plánovací strategie. Tato třída je používána pomocí následujících dvou metod:

- `rescheduleCall()` – Tato metoda zajišťuje spuštění plánovacího algoritmu pro jeden cyklus plánovače. Naplánované procesy jsou uloženy uvnitř třídy strategie a jsou přístupné plánovači pomocí následující metody.
- `getSelectedProcess(int cpuIndex)` – Tato metoda vrací proces naplánovaný pro zvolený interpret.

Třídy odděděné od třídy `ISchedulerStrategy` implementují jednotlivé plánovací strategie popsané v kapitole 4.4.

Algoritmus plánování lze nastavit v konfiguračním souboru uzlu SAN, k dispozici jsou následující strategie:

- `RoundRobinStrategy` – Popsána v kapitole 4.4.3.
- `MultiLevelQueueStrategy` – Popsána v kapitole 4.4.4.
- `PriorityStrategy` – Popsána v kapitole 4.4.6.
- `LoteryStrategy` – Popsána v kapitole 4.4.7.

Třída `StrategyManager` se stará o zavedení zvolené strategie do plánovače.

6.6.9 Optimalizační mezivrstva pro více procesorů

Optimalizační mezivrstva mezi plánovací strategií a zavaděčem má za úkol zajistit optimální rozložení procesů na více interpretrů. Je využito algoritmu popsaného v kapitole 4.5. Implementace mezivrstvy se nachází v metodě `optimalize()` ve třídě rozhraní strategie `ISchedulerStrategy` a je provedena zároveň s voláním plánovací metody `rescheduleCall()`.

6.6.10 Zavaděč

Součást plánovače, která přistupuje k interpretrům a řídí spouštění a pozastavování aplikací podle rozhodnutí plánovací strategie. Zavaděč je implementován v metodě `executeProcess(…)` volané metodou `schedule()`. Přístup k interpretrům je zajištěn pomocí rozhraní `san.interpreter.IInterpreter`.

6.7 Moduly plánovače

Třída plánovače obsahuje funkcionalitu pro zajištění plánování, spouštění a pozastavování procesů. Funkcionalita synchronizačních prostředků je do plánovače dodána pomocí rozšiřujících modulů.

Modul je třída implementující rozhraní `ISchedulerModule`. Toto rozhraní obsahuje dvě metody:

- `checkProcessCall(Process process)` – Tato metoda je volána, pokud interpretr přeruší vykonávání procesu z důvodu volání služby modulu. Určení modulu, kterému volání patří, je zajištěno pomocí stavu procesu. Modul může v této metodě nastavením stavu procesu rozhodnout, zda proces může pokračovat v běhu, nebo je nutné jej pozastavit a naplánovat jiný. Tuto metodu volá pouze interpret.
- `checkAllProcesses(…)` – Tato metoda je spouštěna při každé iteraci plánovače a slouží ke kontrole stavu procesů, které modul blokuje. Tuto metodu volá pouze plánovač.

6.7.1 Rozhraní poskytovaná moduly

Každý modul poskytuje rozhraní, kterým zpřístupňuje své služby uživatelským aplikacím a kapsulím. Tato rozhraní rozšiřují API a CPI. Implementace metod rozhraní je

umístěna ve třídě `san.core.Process`. Volání služeb z těchto metod je zajištěno pomocí schránky (třídy `ConditionRequest` a `MonitorRequest`). Do schránky je vložen identifikátor požadované služby a její argumenty. Určení, kterému modulu je zpráva zaslána, je zajištěno pomocí stavu procesu. Modul je následně zavolán metodou `checkProcessCall(…)` z metody `intepretingEnded(…)` volané interpretrem.

Tento postup byl zvolen z důvodu odstínění plánovače od interpretovaného procesu. Proces nemůže nijak zasahovat do kontextu plánovače. Důvodem je bezpečnost. Pokud by proces havaroval při volání služby uvnitř kontextu plánovače, mohla by být ohrožena činnost celého uzlu. Technika zasílání zpráv tento problém odstraňuje.

Rozhraní poskytnutá moduly jsou umístěna v balíku `san.core.fiber.api` a rozšiřují původní API a CPI. Jedná se o následující rozhraní:

- `ISleepAPI` – Rozhraní pro uspávání procesů.
- `IConditionAPI` – Rozhraní pro podmínkové proměnné.
- `IMonitorAPI` – Rozhraní pro synchronizační monitory.
- `IInfoGetterAPI` – Obsahuje metodu pro vrácení objektu implementujícího rozhraní `IInfoAPI`.
- `IInfoAPI` – Rozhraní pro získání statistiky a konfigurace plánovače. Pro udržení přehlednosti je rozhraní statistiky poskytnuto ve zvláštním objektu mimo API a CPI.

6.7.2 Modul pro správu spánku

Modul `SleepManager` zajišťuje správu uspaných procesů. Procesy jsou vždy uspávány na danou dobu. Dokud nevyprší doba spánku, modul nedovolí naplánování procesu. Uspané procesy jsou ukládány v seznamu a jejich čas probuzení je periodicky kontrolován.

Pro přístup ke službám modulu pro správu spánku slouží rozhraní `ISleepAPI` obsahující následující metody:

- `sleep(long timeout)` – Uspání procesu na daný čas.
- `yield()` – Zřeknutí se zbytku přiděleného časového kvanta.

Výše uvedené metody lze nahradit přetíženými metodami třídy `Thread`, funkčnost přetížených metod je identická.

6.7.3 Modul pro správu podmínkových proměnných

Modul `ConditionManager` zajišťuje mechanizmus podmínkových proměnných. Jako identifikátor podmínkové proměnné bylo použito celého čísla typu `int`. Všechny metody pro práci s podmínkami vyžadují jako argument tento identifikátor. Jednotlivým podmínkám je přiřazena fronta procesů, které nad podmínkou čekají. Fronta zajišťuje FIFO

posloupnost probouzení čekajících procesů. Před prvním použitím je potřeba podmínkovou proměnnou inicializovat.

Služby modulu pro správu podmínkových proměnných jsou přístupné pomocí rozhraní `IConditionAPI`, které poskytuje následující metody:

- `conditionInitialize(int id)` – Metoda pro inicializaci podmínky.
- `conditionDestroy(int id)` – Metoda pro zrušení podmínky.
- `conditionWait(int id)` – Metoda pro čekání na podmínce. Proces volající tuto metodu je zablokován, dokud neobdrží signál.
- `conditionWait(int id, long timeout)` – Metoda pro čekání na podmínce s časovým limitem. Proces volající tuto metodu je zablokován, dokud neobdrží signál nebo dokud není překročen časový limit.
- `conditionSignalOne(int id)` – Metoda pro zaslání signálu jednomu čekajícímu procesu.
- `conditionSignalAll(int id)` – Metoda pro zaslání signálu všem čekajícím procesům.
- `conditionGetWaitingCount(int id)` – Metoda pro vrácení počtu procesů čekajících na dané podmínce.

Synchronizační metody třídy `Object` nejsou v současné verzi plánovače a interpretru podporovány. Volání těchto metod je nyní zablokováno vyhozením výjimky.

6.7.4 Modul pro správu synchronizačních monitorů

Modul `MonitorManager` zajišťuje mechanizmus synchronizace pro monitory. Modul si ukládá ke každému použitému monitoru proces vlastnící monitor, počet zamčení monitoru tímto procesem a frontu čekajících procesů.

Služby modulu pro správu synchronizačních monitorů jsou přístupné pomocí rozhraní `IMonitorAPI`, které poskytuje následující metody:

- `monitorEnter(Object monitor)` – Metoda pro vstup do monitoru.
- `monitorExit(Object monitor)` – Metoda pro opuštění monitoru.
- `monitorTest(Object monitor)` – Metoda pro otestování, zda je monitor volný. Tato metoda však nezajišťuje atomičnost. Následné zabrání monitoru může způsobit zablokování procesu, i když podle předchozího volání této metody byl monitor volný.
- `monitorTestAndEnter(Object monitor)` – Metoda pro atomické otestování monitoru a jeho následné zabrání (TSL).
- `monitorSetSequence(Object[] monitorSequence)` – Metoda pro registraci posloupnosti monitorů, které chce proces zabírat. Sekvence monitorů je potřebná pro činnost modulu prevence uvíznutí.

Místo dvojice metod `monitorEnter(…)` a `monitorExit(…)` je možné použít blok `synchronized (…) {…}`. Funkčnost obou zápisů je identická.

6.7.5 Modul pro detekci uvíznutí

Modul `DeadlockDetector` slouží k nalezení a odstranění uvíznutí. Modul není přímo přístupný plánovači, je zapouzdřený v modulu správce monitorů. Důvodem jsou společné datové struktury. Tento modul pracuje zcela samostatně, a proto nepotřebuje žádné uživatelské rozhraní. V konfiguračním souboru uzlu je možné detekci povolit nebo zakázat a případně nastavit interval spouštění detekce.

Detekce uvíznutí je realizována grafem alokace zdrojů (RAG) popsaným v kapitole 4.8.2.

Operaci detekce uvíznutí je vhodné vzhledem k její výpočetní náročnosti spouštět ve větších intervalech, implicitní hodnota je 1000 ms.

6.7.6 Modul pro prevenci uvíznutí

Modul pro prevenci uvíznutí nemá vlastní třídu a je zakomponován přímo do modulu správce monitorů, protože s ním sdílí datové struktury a velkou část funkcionality.

Prevence uvíznutí je zajištěna pomocí bankéřova algoritmu. Procesy si mohou zaregistrovat posloupnosti monitorů, které chtějí zabírat. K tomuto slouží metoda `monitorSetSequence(Object[] monitorSequence)`.

Bankéřův algoritmus je implementován podle popisu v kapitole 4.8.3.

Nevýhodou tohoto řešení je jeho závislost na správnosti poskytnuté informace jednotlivými procesy. Pokud proces uvede nesprávnou posloupnost monitorů, nemůže bankéřův algoritmus spolehlivě fungovat.

7 Dosažené výsledky

7.1 Ukázková aplikace

Pro ověření funkčnosti implementace a prezentaci nových funkcí uzlu SAN byla vytvořena ukázková aplikace `fiber`.

Spuštěním aplikace bez parametrů pomocí příkazu "`> run fiber`" zadaného do konzole uzlu SAN lze vypsat následující nápovědu.

```
> run fiber
app guid D25BC7B4-4BB4-0A38-D9B0-A02DA435D350
+------------------------------+
| Fiber scheduler demonstration |
+------------------------------+

Usage:
    > run fiber <feature> [thread_count]

Features:
    (S) Sleep ----------- Suspending threads
    (C) Condition ------- Threads waiting on condition
    (M) Monitor --------- Accessing shared context with monitor
    (N) MonitorNo ------- Same as previous, but without monitor
    (D) DeadlockDetect -- Finding monitor deadlocks
    (P) DeadlockPrevent - Preventing monitor deadlocks
    (T) DeadlockTest ---- Acquire monitor with TSL method
    (I) Info ------------ Scheduler information and statistic

Feature real usages:
    (X) PiSingle -------- Separate PI calculating
    (Y) PiMulti --------- Shared PI calculating

>
```

Ukázky lze spustit příkazem "`> run fiber <ukázka> [počet_vláken]`". Každá ukázka má přiřazeno jedno písmeno (uvedené v závorce), kterým je možné ji vyvolat. Počet vláken je volitelný parametr, pokud není uveden, je použita implicitní hodnota 10.

Popis jednotlivých ukázek:

- (S) Sleep – Ukázka uspání vláken na určitý čas.
- (C) Condition – Ukázka použití podmínkových proměnných při řízení vláken.
- (M) Monitor – Ukázka použití monitoru pro přístup ke sdílené proměnné v kritické sekci.
- (N) MonitorNo – Ukázka přístupu ke sdílené proměnné bez synchronizace.
- (D) DeadlockDetect – Ukázka detekce a odstranění uvíznutí při zabírání monitorů.
- (P) DeadlockPrevent – Ukázka předcházení uvíznutí pomocí bankéřova algoritmu.

- (T) DeadlockTest – Ukázka předcházení uvíznutí pomocí TSL přístupu k monitoru.
- (I) Info – Výpis statistiky plánovače.
- (X) PiSingle – Ukázka nesdíleného výpočtu čísla PI pomocí více vláken, každé vlákno počítá svojí vlastní hodnotu PI.
- (Y) PiMulti – Ukázka sdíleného výpočtu čísla PI pomocí více vláken, vlákna společně počítají jednu hodnotu čísla PI a komunikují spolu pomocí sdílených proměnných.

Každé ukázce odpovídá jedna třída v balíku aplikace. Základem každé ukázky je abstraktní třída `IFeature`, která zajišťuje spouštění a ukončování pracovních procesů podle schématu farmer-worker.

7.2 Praktická měření

Pro určení výkonu výsledného řešení bylo provedeno několik testů. Testy mají za úkol prověřit implementaci fiber vláken a plánovače z hlediska paměťových nároků, rychlosti odezvy plánovače a rychlosti interpretace fiber vláken.

Pro porovnání nám poslouží původní implementace uzlu SAN, ze které tato práce vychází. Měření bude zahrnovat celý uzel SAN, tedy paměť a vlákna všech komponent. Porovnáním původní a naší implementace získáme potřebnou představu o výkonu a nárocích nového řešení plánovače.

Aplikace `fiber` není zcela přenositelná na původní uzel SAN, protože některé její části využívají nové rozhraní plánovače. Proto byl pro testování vybrán nesdílený výpočet čísla PI, který nevyžaduje žádné nové rozhraní a jeho implementace je na původní uzel SAN přenositelná.

Testy byly provedeny na osobním počítači s následující konfigurací (*Tab. 01*).

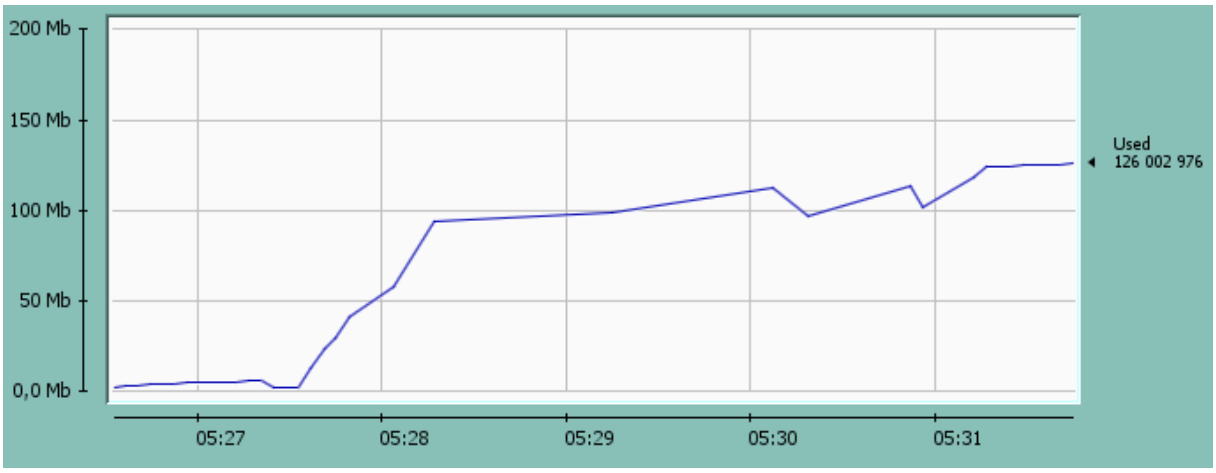
Procesor	AMD Athlon 64 X2 4400+ (2,3 GHz)
Operační paměť	3 GB DDR2 (800 MHz)
Operační systém	Windows XP Pro SP3 x32
Verze JRE	1.6.0_21-b07

Tab. 1 – Konfigurace testovacího počítače.

7.2.1 Paměťové nároky

Měření paměťových nároků uzlu SAN bylo provedeno pomocí nástroje jConsole, který je standardní součástí JDK. Nástroj jConsole umožňuje u spuštěné Java aplikace sledovat množství alokované paměti, počet aktivních vláken a využití procesoru.

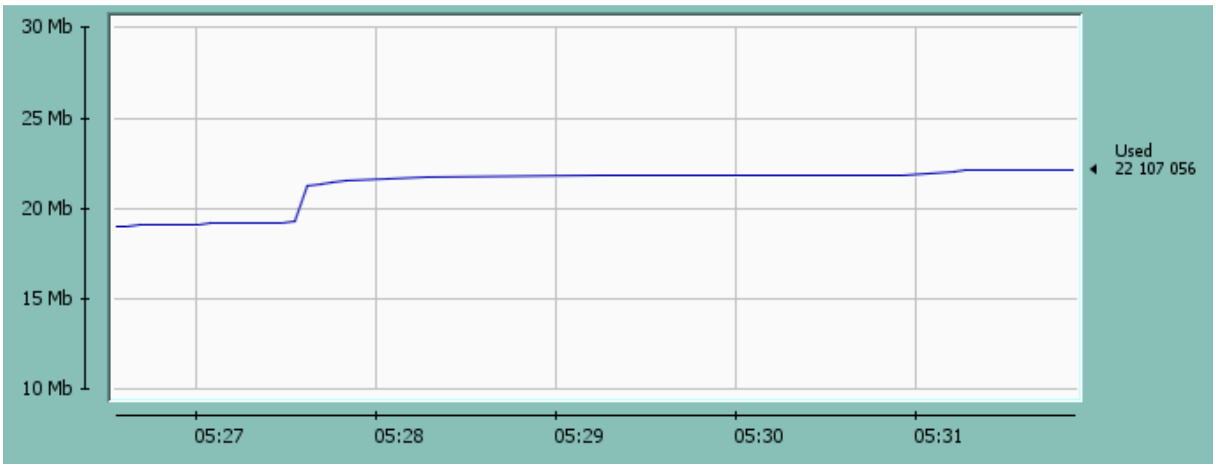
Pro měření byla zvolena aplikace `fiber`, konkrétně nesdílený výpočet čísla PI pro 100 vláken. Aplikace byla spuštěna příkazem“`> run fiber X 100`“. Na uzlu neběžela žádná jiná aplikace nebo kapsule.



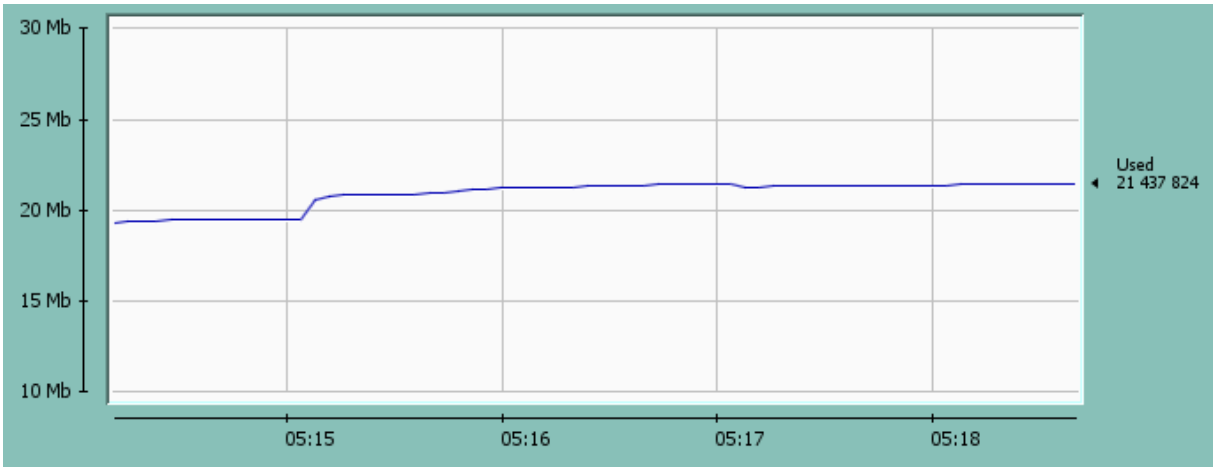
Obr. 06 – Velikost alokované paměti na haldě pro původní uzel SAN.

Obr. 07 – Velikost alokované paměti na haldě pro fiber uzel SAN.

Halda je část paměti virtuálního stroje Java, kde se ukládají vytvořené objekty. Z grafů alokované paměti na haldě (*Obr. 06* a *Obr. 07*) je patrné, že použitím vlastní implementace fiber vláken došlo k výrazné úspoře paměti. Protože pro každý proces není vytvořena vlastní instance interpretru, dochází k výraznému omezení alokace nových objektů, v tomto případě téměř trojnásobně.

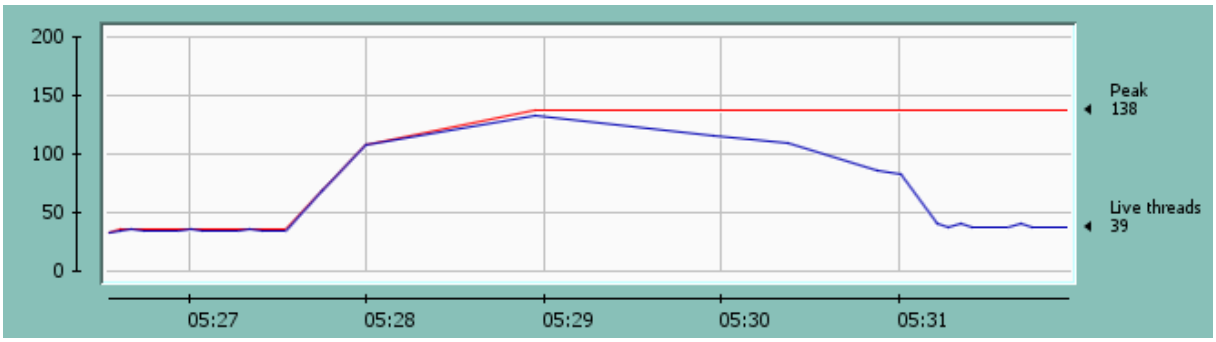


Obr. 08 – Velikost alokované paměti mimo haldu pro původní uzel SAN.

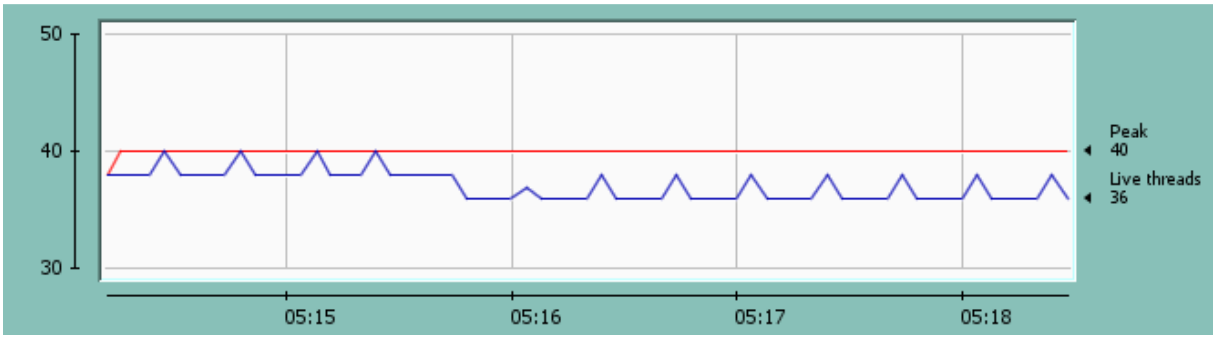


Obr. 09 – Velikost alokované paměti mimo haldu pro fiber uzel SAN.

Paměť mimo haldu slouží k ukládání načtených tříd a ostatních metadat. Grafy alokované paměti mimo haldu (*Obr. 08* a *Obr. 09*) jsou téměř identické, protože naše implementace nijak neměnila přístup k načítání tříd.



Obr. 10 – Počet aktivních vláken pro původní uzel SAN.



Obr. 11 – Počet aktivních vláken pro fiber uzel SAN.

Z grafů průběhu počtu aktivních vláken (*Obr. 10* a *Obr. 11*) je jasně vidět, že původní verze uzlu SAN spouštěla procesy v nových vláknech. Fiber verze uzlu SAN nespouštěla v průběhu výpočtu žádná nová vlákna, protože vlákna interpretrů byla vytvořena po startu uzlu a procesy se o ně jen střídaly.

Drobné periodické výchylky v počtu vláken jsou způsobeny interními časovači JRE pro grafické rozhraní uzlu nebo pro přístup k síti.

7.2.2 Rychlost odezvy plánovače

Pro otestování rychlosti odezvy plánovače bude použita aplikace `ping`. Tato aplikace slouží k otestování dosažitelnosti jiných uzlů.

Činnost aplikace `ping` lze popsat následovně:

1) Naplánování a spuštění aplikace na lokálním uzlu.
2) Injektování kapsule.
3) Odeslání kapsule na vzdálený uzel.
4) Naplánování a spuštění kapsule na vzdáleném uzlu.
5) Určení, zda se kapsule nachází na cílovém uzlu.
6) Odeslání kapsule zpět na lokální uzel.
7) Naplánování a spuštění kapsule na lokálním uzlu.
8) Předání dat od kapsule do aplikace.
9) Vyhodnocení doručených dat v aplikaci.

Činnost aplikace `ping` vyžaduje spolupráci všech komponent uzlu SAN, obzvláště plánovače a interpretru. Proto je aplikace `ping` ideální pro testování rychlosti odezvy plánovače a interpretru.

Celkem budou provedeny 3 sady testů:

- Pro jeden uzel SAN – Bude využito zpětnovazební rozhraní, uzel pošle požadavek `ping` sám sobě a také sám sobě pošle odpověď.
- Pro dva uzly SAN – První uzel pošle požadavek `ping` druhému uzlu, který nazpět zašle odpověď.

- Pro tři uzly SAN – První uzel pošle požadavek `ping` přes druhý uzel třetímu uzlu, který nazpět zašle odpověď opět přes druhý uzel. Při průchodu druhým uzlem je kapsule také spouštěna.

Uzly jsou označený čísly 1, 2 a 3. Aplikace `ping` bude spouštěna vždy z uzlu číslo 1. V následujících tabulkách jsou zaznamenány 3 časy – doba potřebná pro doručení kapsule na cílový uzel, doba potřebná pro doručení kapsule zpět a celková doba od odeslání požadavku do doručení a vyhodnocení odpovědi.

Pro potřeby testu byla aplikace `ping` upravena tak, aby využívala nové možnosti synchronizace, které náš plánovač poskytuje. Použita byla podmínková proměnná pro probuzení aplikace po doručení odpovědi.

Dále byla vytvořena zcela nová implementace aplikace `ping` jménem `fiberping`. Kromě výše zmíněného použití podmínkové proměnné byla aplikace od základu upravena a optimalizována. Tato verze není v následujícím testu použita, protože výsledek by nebyl porovnatelný s původní verzí.

Původní uzel SAN (1)	**Měření odezvy aplikace `ping` [ms]**										**Průměrná hodnota**	**Směrodatná odchylka**
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**		
1 -> 1	31	94	31	47	31	32	63	46	31	31	43,7	19,7
1 <- 1	110	46	109	93	109	109	93	94	156	110	102,9	25,6
1 <-> 1	141	140	140	140	140	141	156	140	187	141	146,6	14,2

Tab. 02 – Rychlost odezvy pro 1 původní uzel SAN.

Fiber uzel SAN (1)	**Měření odezvy aplikace `ping` [ms]**										**Průměrná hodnota**	**Směrodatná odchylka**
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**		
1 -> 1	63	63	62	62	62	157	47	156	172	156	100,0	49,6
1 <- 1	156	156	157	266	63	140	140	141	156	47	142,2	56,1
1 <-> 1	219	219	219	328	125	297	187	297	328	203	242,2	63,8

Tab. 03 – Rychlost odezvy pro 1 fiber uzel SAN.

Původní uzel SAN (2)	**Měření odezvy aplikace `ping` [ms]**										**Průměrná hodnota**	**Směrodatná odchylka**
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**		
1 -> 2	46	47	47	46	125	47	63	109	62	63	65,5	26,9
1 <- 2	32	47	47	32	47	78	31	32	47	31	42,4	13,9
1 <-> 2	78	94	94	78	172	125	94	141	109	94	107,9	28,4

Tab. 04 – Rychlost odezvy pro 2 původní uzly SAN.

Fiber uzel SAN (2)	Měření odezvy aplikace `ping` [ms]										Průměrná hodnota	Směrodatná odchylka
	1	2	3	4	5	6	7	8	9	10		
1 -> 2	46	47	125	62	78	47	78	63	63	63	67,2	22,2
1 <- 2	63	156	47	47	47	78	47	47	46	62	64,0	32,3
1 <-> 2	109	203	172	109	125	125	125	110	109	125	131,2	29,8

Tab. 05 – Rychlost odezvy pro 2 fiber uzly SAN.

Původní uzel SAN (3)	Měření odezvy aplikace `ping` [ms]										Průměrná hodnota	Směrodatná odchylka
	1	2	3	4	5	6	7	8	9	10		
1 -> 2 -> 3	62	94	62	47	62	47	63	62	63	94	65,6	15,4
1 <- 2 <- 3	78	47	47	78	32	47	62	47	31	47	51,6	15,5
1 <-> 2 <-> 3	140	141	109	125	94	94	125	109	94	141	117,2	18,8

Tab. 06 – Rychlost odezvy pro 3 původní uzly SAN.

Fiber uzel SAN (3)	Měření odezvy aplikace `ping` [ms]										Průměrná hodnota	Směrodatná odchylka
	1	2	3	4	5	6	7	8	9	10		
1 -> 2 -> 3	62	79	62	62	62	78	78	62	78	78	70,1	8,1
1 <- 2 <- 3	47	62	47	63	47	63	47	63	47	47	53,3	7,7
1 <-> 2 <-> 3	109	141	109	125	109	141	125	125	125	125	123,4	11,2

Tab. 07 – Rychlost odezvy pro 3 fiber uzly SAN.

V případě dvou a tří uzlů je rozdíl v odezvě plánovače průměrně 15 ms (+10%). Náš plánovač nikdy nemůže být rychlejší než původní řešení. Je potřeba navíc před spuštěním každé kapsule provést její naplánování. Při doručování dat z kapsule do aplikace nemůže být spuštěna doručovací metoda přímo, ale musí být obalena novým procesem a opět nejdříve naplánována. Za těchto okolností je zpomalení o 10% více než přijatelné.

V případě jednoho uzlu (zpětnovazební rozhraní) je průměrný rozdíl odezvy roven 100 ms (+70%). Tento rozdíl je zřejmě způsoben architekturou plánovače. V Původním řešení je po doručení kapsule okamžitě jádrem spuštěna metoda pro doručení dat aplikaci. V našem případě však musíme volání této metody nejdříve obalit novým procesem a tento proces naplánovat. Zde je naše řešení oproti původnímu v nevýhodě. V předchozích dvou případech se tento neduh musel také projevit, ale díky zapojení více uzlů a skutečnému odesílání a přijímání kapsule se neprojevil až tak výrazně.

7.2.3 Rychlost interpretace

Pro otestování rychlosti interpretace fiber vláken bude opět použita aplikace `fiber` a její nesdílený výpočet čísla PI. Algoritmus výpočtu je naznačen na následujícím fragmentu kódu. Aplikace bude spuštěna pro 10 vláken příkazem "`> run fiber X 10`".

```
double calculatePI() {
    double pi = 0.0;
    double segment;
    for (int i = 0; i <= ITERATION_COUNT; i++) {
        segment = 4.0 / (2 * i + 1);
        if (i % 2 != 0) {
            segment = -segment;
        }
        pi += segment;
    }
    return pi;
}
```

Výpočet čísla PI probíhá v jednom for cyklu a bez volání dalších metod. Použité jazykové prostředky pro zapsání algoritmu výpočtu jsou for cyklus, porovnání, podmínka, lokální proměnné, sčítání, odčítání, násobení, dělení, modulo a přiřazení. Celý výpočet probíhá zcela v režii třídy `FiberInterpreterBytecode`. Během výpočtu nedochází k žádnému volání metod, načítání tříd, hledání bytecode pro metody, alokace paměti nebo vytváření nových objektů.

To znamená, že není do výpočtu zahrnuto použití nástroje reflexe nebo načítání tříd pomocí class loaderu. Výpočet se skládá jen z jednoduchých instrukcí manipulujících s instrukčním ukazatelem, lokálními proměnnými a datovým zásobníkem. Proto budou výsledky odrážet reálný výpočetní výkon interpretru.

Do testu byl kromě původního a fiber uzlu SAN zařazen také Bean Shell interpretr. Pro určení zpomalení rychlosti výpočtu vlivem interpretace byl algoritmus také spuštěn přímo jako program pro JRE. Výpočet byl proveden v 10 vláknech s konstantním počtem iterací. Výsledek testu je uveden v následující tabulce (*Tab. 08*).

Interpretr	**Měření doby běhu aplikace `fiber X 10` [ms]**					**Průměrná hodnota**	**Směrodatná odchylka**
	1	**2**	**3**	**4**	**5**		
Fiber SAN	279578	271734	255266	275266	255594	267487,6	10154,2
Původní SAN	25141	25110	25250	25281	25391	25234,6	101,1
Bean Shell	652312	641625	696766	650094	658172	659793,8	19234,1
JRE	531	485	468	515	484	496,6	22,9
JRE bez JIT	1344	1328	1422	1375	1391	1372	33,4

Tab. 08 – Rychlost výpočtu čísla PI v závislosti na použitém interpretru.

Z výsledku testu je bohužel patrné, že rychlost výpočtu ve fiber vláknech se 10 krát zhoršila v porovnání s původní implementací uzlu SAN. Proč k tomuto zhoršení došlo? Původní SAN interpretr vykonával postupně jednotlivé instrukce. Upravený SAN fiber interpretr zkontroluje, zda nebyl naplánován jiný proces, zkontroluje kontext vykonávání procesu a vykoná jednu instrukci. Důvodem zpomalení je zde zřejmě režie před vykonáním samotné instrukce.

Pokusme se nyní režii zmenšit tím, že po kontrole naplánování nového procesu vykonáme více než jednu instrukci. Výsledek tohoto rozšíření předchozího testu je uveden v následující tabulce (*Tab. 09*).

Interpretr	Měření doby běhu aplikace `fiber X 10` [ms]					Průměrná hodnota	Směrodatná odchylka
	1	2	3	4	5		
Fiber SAN 1 instrukce za iteraci	279578	271734	255266	275266	255594	267487,6	10154,2
Fiber SAN 10 instrukcí za iteraci	209078	202375	204047	203937	203813	204650,0	2295,9

Tab. 09 – Rychlost výpočtu čísla PI v závislosti na počtu vykonaných instrukcí za iteraci.

Z výsledku tohoto testu je patrné, že zpomalení bylo skutečně způsobeno dodatečnou režií. Zvýšením počtu instrukcí z 1 na 10 za jednu iteraci interpretru jsme dosáhli 25% urychlení. Tímto jsme zmenšili množství režie potřebné na provedení 1 instrukce.

Pokud bychom dále zvyšovali počet instrukcí na jednu iteraci, zřejmě bychom se ještě více přiblížili výkonu původního SAN interpretru. Ovšem nárůst výkonu by byl vykoupen snížením granularity vykonávání procesů, kterou vyžaduje plánovač. Interpretry by reagovaly na naplánované procesy se zpožděním a celé plánování by ztratilo smysl. Plánovač by nemohl využít veškerého výkonu dostupných procesorů a zajistit rychlou odezvu systému.

Navíc je zde ještě režie na kontrolu kontextu vykonávaného procesu, která musí být provedena před každou vykonanou instrukcí. Tuto kontrolu nelze vynechat. Je vždy potřeba zkontrolovat, zda v zásobníku volání metod nepřibyl nový rámec.

7.3 Vyhodnocení výsledků

Naším cílem bylo zmenšit paměťové nároky interpretace, zajistit co nejrychlejší odezvu plánovače a co nejrychlejší interpretaci.

V případě paměťových nároků jsme byli úspěšní, v daném modelovém případě se 100 spuštěnými vlákny se podařilo docílit téměř trojnásobné úspory paměti. Během vykonávání taktéž již není potřeba spouštět žádná nová vlákna JRE.

Pokud se na výsledné řešení podíváme z hlediska rychlosti odezvy, podařilo se zajistit odezvu jen o 10% horší, než v původní implementaci. Vzhledem k tomu, že řešení našeho plánovače bude vždy kvůli jeho režii pomalejší, je výsledek přijatelný.

Z hlediska rychlosti vykonávání instrukcí došlo k výraznému zpomalení. V porovnání s původním SAN interpretrem je naše řešení téměř 10 krát pomalejší. Zpomalení je způsobeno režií potřebnou pro kontrolu nad prováděním jednotlivých instrukcí. Neefektivní interpretování aplikací je bohužel limitujícím faktorem pro využití projektu SAN jako výpočetního prostředí pro distribuované výpočty.

8 Závěr

Tato práce pojednává o konceptu aktivních sítí. Výzkum problematiky aktivních sítí probíhá od roku 1995 a doposud se nepodařilo uspokojivě vyřešit otázku bezpečnosti a výkonnosti. Na katedře Informatiky a výpočetní techniky se od roku 2007 vyvíjí vlastní implementace aktivní sítě – projekt Smart Active Node, která si klade za cíl výše zmíněné nedostatky aktivních sítí vyřešit.

Ačkoliv se v posledních letech objevuje trend vytváření levných, jednoúčelových a kompaktních zařízení s nízkým výkonem a spotřebou, dostupný výpočetní výkon počítačové techniky neustále roste. Otázkou je však, jak jej smysluplně využít. Aktivní sítě mohou být dobrý způsob, jak tento výpočetní výkon v budoucnu využít a zajistit tak novou úroveň služeb poskytovaných počítačovými sítěmi.

8.1 Dosažené cíle

Cílem této práce bylo navrhnout, implementovat a ověřit funkčnost vlastního systému fiber vláken a jejich plánování pro uzel SAN. Důvodů pro přechod od nativních vláken JRE k vlastní implementaci fiber vláken bylo několik.

Prvním důvodem bylo zajištění požadované úrovně zabezpečení spuštěných uživatelských aplikací a kapsulí, kterou nativní vlákna JRE neposkytovala. Díky použití vlastního interpretru je možné spuštěné aplikace a kapsule důsledně kontrolovat a zajistit tak požadovanou míru zabezpečení.

Druhým důvodem byla potřeba umožnit vlastní řízení priority při vykonávání aplikací a kapsulí. Nyní je možné aplikacím a kapsulím přiřazovat prioritní třídy a odlišit tak výpočetní úkoly (vyžadující dlouhodobý výpočet) od interaktivních (vyžadujících rychlou a krátkou odezvu). Pro tyto potřeby byl implementován vlastní plánovač fiber vláken, který nyní zajišťuje přidělování procesorového času.

Třetím důvodem byla snaha maximálně využít výkonu víceprocesorových počítačů. Pro tyto účely byl interpretr a plánovač upraven tak, aby bylo možné plánovat a vykonávat více uživatelských aplikací a kapsulí souběžně.

Čtvrtým důvodem bylo snížení nároků uzlu SAN na systémové zdroje. Procesy již nejsou spouštěny v nových vláknech a pro každý proces není vytvořena vlastní instance interpretru.

Posledním důvodem byla snaha rozšířit služby poskytované uzlem SAN o synchronizační prostředky vytvořené na míru potřebám aktivních sítí. Díky vlastnímu plánovači bylo možné implementovat synchronizaci pomocí podmínkových proměnných a monitorů s potřebnou mírou zabezpečení. Navíc pro tyto synchronizační prostředky byly také implementovány mechanizmy pro detekci a předcházení uvíznutí.

Nové vlastnosti, funkcionalita a zabezpečení uzlu SAN však byly vykoupeny snížením jeho výpočetního výkonu. Nezanedbatelnou část výkonu spotřebovává režie na provoz a řízení těchto nových služeb.

8.2 Budoucí práce

V současné době je v projektu SAN nejslabším místem jeho Java-In-Java interpretr. Rychlost vykonávání výpočetních úloh není dostatečná a neumožňuje rozumné nasazení uzlu SAN jako distribuovaného výpočetního prostředí. Do budoucna se plánuje vytvoření nové implementace uzlu SAN – projekt SAN++. Tento projekt bude vycházet z poznatků získaných v projektu SAN a pokusí se vyřešit problém s rychlostí interpretace uživatelských aplikací a kapsulí. Je zamýšleno využít překladu bytecode interpretovaných uživatelských aplikací a kapsulí přímo do instrukční sady procesoru. Podobným způsobem pracuje kompilátor JIT virtuálního stroje Java.

V projektu SAN++ se předpokládá převzetí plánovače a synchronizačních prostředků vytvořených v rámci této diplomové práce.

9 Přehled zkratek

SAN – Smart Active Node
DARPA – Defense Advanced Research Projects Agency
TCP – Transmission Control Protocol
UDP – User Datagram Protocol
IP – Internet Protocol
JVM – Java Virtual Machine
JRE – Java Runtime Environment
JDK – Java Development Kit
JIT – Just In Time
BSH – Bean Shell
QoS – Quality of Service
SoC – System on Chip
ATM – Asynchronous Transfer Mode
POSIX – Portable Operating System Interface
EE – Execution Environment
SJF – Shortest Job First
SRT – Shortest Remaining Time
FCFS – First Come, First Served
FIFO – First In, First Out
RR – Round Robin
MLQ – Multi-Level Queue
MLFQ – Multi-Level Feedback Queue
API – Application Programming Interface
CPI – Capsule Programming Interface
TSL – Test and Set Lock
BFS – Breadth First Search

10 Použitá literatura

[DPP98] DECASPER, D. – PARULKAR, G. – PLATTNER, B. *A Scalable, High Performance Active Network Node*, IEEE Network conference, 1998, [citováno 1. 5. 2011], <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.48.8251&rep=rep1&type=pdf>.

[EMD88] ENGELMORE, R. – MORGAN, A. – CORKILL, D. *Blackboard Systems*, Addison-Wesley, 1988, [citováno 1. 5. 2011], <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.137.7801&rep=rep1&type=pdf>.

[KMHW98] KULKARNI, A. – MINDEN, G. – HILL, R. – WIJATA, Y. *Implementation of a Prototype Active Network*, Open Architectures and Network Programming conference, 1998, [citováno 1. 5. 2011], <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.50.2872&rep=rep1&type=pdf>.

[Kou04] KOUTNÝ, T. *Reference Implementation of Grade32 AN Server*, 2004, [citováno 1. 5. 2011], <http://www.kiv.zcu.cz/~txkoutny/download/grade32.zip>.

[MBZC00] MEGURU, S. – BHATTACHARJEE, S. – ZEGURA, E. – CALVERT, K. *Bowman: A Node OS for Active Network*, INFOCOM conference, 2000, [citováno 1. 5. 2011], <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.40.9701&rep=rep1&type=pdf>.

[Nyg98] NYGREN, E. *The Design and Implementation of a High-Performance Active Network Node*, 1998, [citováno 1. 5. 2011], <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.50.2872&rep=rep1&type=pdf>.

[Rej08] REJDA, M. *Server aktivní sítě*, Plzeň, 2008, Diplomová práce na Fakultě aplikovaných věd Západočeské univerzity v Plzni na katedře Informatiky a výpočetní techniky. Vedoucí diplomové práce Ing. Tomáš Koutný, Ph.D.

[SGG00] SILBERSCHATZ, A. – GALVIN, P. – GAGNE, G. *Applied operating system concepts*, John Wiley & Sons, 2000, First edition, ISBN 0-471-36508-4.

[SK10] SYKORA, J. – KOUTNÝ, T. *Enhancing Performance of Networking Applications by IP Tunneling through Active Networks*, International Conference on Networks, France, 2010, [citováno 1. 5. 2011], <http://www.computer.org/portal/web/csdl/doi/10.1109/ICN.2010.8>.

[Ste09] ŠTĚPÁNEK, P. *Distribuce kódu v aktivních sítích*, Plzeň, 2009, Diplomová práce na Fakultě aplikovaných věd Západočeské univerzity v Plzni na katedře Informatiky a výpočetní techniky. Vedoucí diplomové práce Ing. Tomáš Koutný, Ph.D.

[Syr09] SYROVÁTKA, J. *Interpretace kódu v Aktivních sítích*, Plzeň, 2009, Diplomová práce na Fakultě aplikovaných věd Západočeské univerzity v Plzni na katedře Informatiky a výpočetní techniky. Vedoucí diplomové práce Ing. Tomáš Koutný, Ph.D.

[TW02] TENNEHOUSE, D. – WETHERALL, D. *Towards an Active Network Architecture*, DARPA Active Networks Conference, 2002, [citováno 1. 5. 2011], <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.49.9823&rep=rep1&type=pdf>.

[WGT98] WETHERALL, D. – GUTTAG, J. – TENNENHOUSE, D. *ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols*, IEEE OPENARCH conference, San Francisco, 1998, [citováno 1. 5. 2011], <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.35.6064&rep=rep1&type=pdf>.

11 Příloha A – Statistika plánovače

Pro sledování aktuální činnosti plánovače, jeho statistik a nastavení je k dispozici rozhraní `IInfoAPI`. Toto rozhraní poskytuje následující informace:

- Jméno uzlu.
- Počet spuštěných, dokončených a zastavených aplikací, kapsulí a speciálních volání.
- Čas a datum spuštění uzlu SAN.
- Čas běhu uzlu SAN.
- Počet fyzických procesorů počítače, na kterém je uzel SAN spuštěn.
- Počet paralelně spuštěných interpretrů (implicitně stejná hodnota jako fyzický počet procesorů).
- Nejvyšší možný počet souběžně běžících kapsulí (implicitně 4).
- Interval plánování (velikost časového kvanta, implicitně 100 ms).
- Jméno použité plánovací strategie (implicitně prioritní plánování).
- Povolení detektoru uvíznutí (implicitně povolen).
- Interval provádění detekce uvíznutí (implicitně 1000 ms).

Předchozí rozhraní je přístupné také pomocí informačního HTTP serveru. Číslo portu je implicitně 47470. Pro snadnější orientaci je port vždy vypsán na standardní výstup. Pro zobrazení informační stránky uzlu je potřeba zadat do adresního řádku webového prohlížeče adresu HTTP serveru (například `http://localhost:47470/`).

12 Příloha B – Vytváření záznamů o činnosti plánovače

Pro vytváření záznamů o činnosti a provozu (logování) plánovače a interpretru je nasazena knihovna Log4j. Jsou použity tři úrovně důležitosti výpisů:

- `DEBUG` – Vypisovány jsou informace o všech dílčích činnostech plánovače a interpretru. Zapnutí této volby značně zpomaluje běh uzlu a je vhodné jen pro vývoj a ladění.
- `WARN` – Tato úroveň výpisů obsahuje informace o chybách při interpretaci uživatelských aplikací a kapsulí. Pokud uživatelská aplikace nebo kapsule skončí neošetřenou výjimkou nebo dojde k uvíznutí, je chyba vypsána touto úrovní výpisů. Výskyt této chyby je tedy podmíněn chybovostí uživatelských aplikací a kapsulí. Chyby této úrovně neovlivňují chod uzlu.
- `ERROR` – Tato úroveň poskytuje informace o interních chybách plánovače a interpretru. Zachytávány jsou všechny neošetřené výjimky, které mohou nastat ve vláknech plánovače nebo interpretru. Chyby této úrovně přímo ovlivňují chod uzlu.

Nastavení úrovně výpisu je možné provést zvlášť pro plánovač a interpretr. Změnu v úrovni výpisů lze provést úpravou atributů ve třídě `FiberLoggerFactory`. Výpis záznamů je prováděn na standardní výstup a do souboru.

13 Příloha C – Konfigurace plánovače

Konfigurace uzlu SAN je zajištěna pomocí XML souboru. V souboru je možné nastavit název uzlu, síťová rozhraní, směrování, typ uživatelské konzole a nastavení úložiště kódu. Konfiguraci bylo potřeba rozšířit o nastavení vlastností plánovače.

Syntaxe konfigurace plánovače je ukázána na následujícím příkladu. Použité hodnoty jsou zároveň hodnotami implicitními. Pokud některý parametr není uveden, je použita jeho implicitní hodnota.

```
<?xml version="1.0" encoding="UTF-8"?>
<node name="SAN Node 1">
    <scheduler virtualCPUCount="2"
               maxCapsulesRunning="4"
               schedulerSleepInterval="100"
               schedulerStrategy="PriorityStrategy"
               deadlockDetectorEnabled="true"
               deadlockDetectorSleepInterval="1000"
               fiberInfoHttpPort="47470"
    />
</node>
```

Význam jednotlivých parametrů:

- `name` – Název uzlu.
- `virtualCPUCount` – Počet paralelně spuštěných interpretrů (implicitně stejná hodnota jako fyzický počet procesorů).
- `maxCapsulesRunning` – Nejvyšší možný počet souběžně běžících kapsulí.
- `schedulerSleepInterval` – Interval plánování (velikost časového kvanta).
- `schedulerStrategy` – Jméno použité plánovací strategie.
- `deadlockDetectorEnabled` – Povolení detektoru uvíznutí.
- `deadlockDetectorSleepInterval` – Interval provádění detekce uvíznutí.
- `fiberInfoHttpPort` – Port informačního HTTP serveru.

14 Příloha D – Popis provedených změn v projektu SAN

Vlastní implementace byla rozdělena do dvou hlavních celků:

- `san.core.fiber` – Balík obsahující implementaci plánovače, plánovacích strategií, synchronizačních modulů a definice nových rozhraní.
- `san.interpreter.sanint.fiber` – Balík obsahující implementaci fiber vláken pro interpretr SAN.

Implementace si vyžádala změny následujících původních tříd:

- `san.core.Process` – Rozšíření počtu stavů procesu, přidání kontextu vykonávání vlákna, zasílání zpráv, implementace nových rozhraní.
- `san.core.ProcessApplication` – Úprava doručování dat z kapsule do aplikace (nutnost volání doručovací metody naplánovat, speciální volání).
- `san.core.IApplicationAPI` – Rozšíření rozhraní.
- `san.core.ICapsuleAPI` – Rozšíření rozhraní.
- `san.core.IScheduler` – Rozšíření rozhraní plánovače o metodu pro spouštění speciálních procesů a metodu pro získání objektu statistiky.
- `san.core.Kernel` – Zavedení nového plánovače, načtení konfigurace.
- `san.data.ReceiveDataListener` – Rozšíření doručovací metody o rozhraní uzlu (pro použití synchronizace v doručovací metodě, nutnost provést změnu všech aplikací využívajících doručovací rozhraní).
- `san.interpreter.sanint.classLoading.Class` – Rozšíření metody `runMethod(…)` o argument reference na aktuálně vykonávaný proces.
- `san.interpreter.sanint.classLoading.HardClass` – Úprava třídy pro podporu fiber vláken, volání nových metod vkládáním do zásobníku volání, přetížení přístupu ke statickým atributům třídy, přetížení volání synchronizačních metod.
- `san.interpreter.sanint.classLoading.ClassManager` – Úprava třídy pro podporu fiber vláken, volání inicializace nově načtené třídy vkládáním do zásobníku volání, ošetření přístupu ke statickým atributům a inicializace třídy `java.lang.System` pomocí zásobníku volání.
- `san.interpreter.InterpretManager` – Zavedení podpory pro více instancí interpretru.
- `san.interpreter.IInterpretingContext` – Vytvoření rozhraní pro správu kontextu vykonávání jednotlivých procesů.

Původní třídy `Scheduler` a `SanInterpreter` nejsou v aktuální implementaci uzlu SAN nadále podporovány a byly nahrazeny novými třídami `FiberScheduler` a `SanFiberInterpreter`.