VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

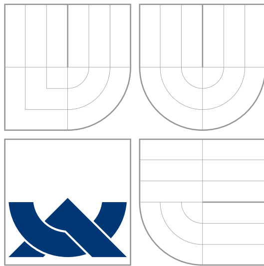
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

VYUŽITÍ TECHNOLOGIE GCD PRO POTŘEBY ANTIVIROVÉHO SOFTWARU

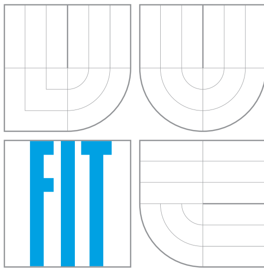
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. MICHAL ŠVEC
AUTHOR

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

VYUŽITÍ TECHNOLOGIE GCD PRO POTŘEBY ANTIVIROVÉHO SOFTWARU

APPLICABILITY OF GCD TECHNOLOGY TO ANTIVIRUS SOFTWARE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. MICHAL ŠVEC
AUTHOR

VEDOUCÍ PRÁCE Ing. RADEK KOČÍ, Ph.D.
SUPERVISOR

BRNO 2011

Abstrakt

Hlavním účelem této práce je popsat technologii Grand Central Dispatch. Ta slouží k paralelnímu programování. Nejdříve je tedy popsána historie a základy paralelního programování. Na to navazuje popis knihovny spolu s příklady a test výkonnosti. Využití je demonstrováno na jednoduchém HTTP serveru, který je napojen na antivirový software AVG.

Abstract

This master's thesis describes Grand Central Dispatch technology. It is technology for parallel computing and task processing. It also describes history and techniques of parallel computing. As an example stand simple HTTP server with 4 methods of request parallel processing. Server is connected to AVG antivirus software and check each request for viruses.

Klíčová slova

Grand Central Dispatch, paralelismus, vlákna, HTTP, server.

Keywords

Grand Central Dispatch, paralelism, threads, HTTP, server.

Citace

Michal Švec: Využití technologie GCD pro potřeby antivirového softwaru, diplomová práce, Brno, FIT VUT v Brně, 2011

Využití technologie GCD pro potřeby antivirového softwaru

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Radka Kočího, Ph.D.

....
Michal Švec
24. května 2011

Poděkování

Děkuji panu Ing. Radku Kočímu, Ph.D. a panu Ing. Jaroslavu Suchánkovi za poskytnuté konzultace a odbornou pomoc. Dále děkuji mým rodičům a přítelkyni za podporu při vypracovávání práce.

© Michal Švec, 2011.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **3**

2 Paralelní výpočty **4**
2.1 Historie paralelního programování 4
2.2 Flynnova taxonomie 5
2.3 Typy paralelismu 6
2.3.1 Aritmetický a bitový stupeň 6
2.3.2 Instrukční stupeň 6
2.3.3 Datový paralelismus 6
2.3.4 Funkční paralelismus 6
2.4 Modely paralelního programování 6
2.4.1 Procesy 6
2.4.2 Vlákna 7
2.4.3 OpenMP 8
2.4.4 Open MPI 8

3 Grand Central Dispatch **10**
3.1 Úvod 11
3.2 Bloky 11
3.3 Fronty 13
3.3.1 Sériové fronty 13
3.3.2 Paralelní fronty 14
3.3.3 Hlavní fronta 14
3.3.4 Operace s frontou 15
3.3.5 Paralelní zpracování cyklu 16
3.4 Skupiny 16
3.5 Semafory 17
3.6 Zdroje 17
3.6.1 Vytvoření zdroje 18
3.6.2 Nastavení obsluhy zdroje 19
3.6.3 Zrušení zdroje 20
3.6.4 Další operace se zdroji 20
3.7 Podpora GCD v operačních systémech 20
3.8 Přepis existujících aplikací do GCD 21
3.9 Praktické využití GCD 21
3.10 Výkonnost 22
3.10.1 Způsob měření 22
3.10.2 Výsledky synchronního měření 24

3.11 Zhodnocení knihovny 27

4 HTTP servery **28**
4.1 Apache 28
4.2 HTTP protokol 28
4.3 URL 29
4.4 HTTP Metody 29
4.5 Rozdíl mezi HTTP 1.0 a 1.1 30
4.5.1 Rozšiřitelnost 30
4.5.2 Cache 30
4.5.3 Přenos dat 31
4.5.4 Komprese přenášených dat 31
4.5.5 Jméno serveru 32
4.5.6 Další rozdíly 32

5 Antivirový software AVG **33**
5.1 Metody detekce 33
5.2 Architektura 34
5.3 Využití GCD 34

6 Implementace HTTP serveru **35**
6.1 Metody zpracování požadavků 35
6.2 Architektura aplikace 36
6.3 Počítadlo požadavků 37
6.4 Testování serveru 37
6.5 Antivirová kontrola souborů 38
6.6 Nastavení serveru 39

7 Závěr **40**

A Obsah DVD **43**

Kapitola 1

Úvod

Paralelní programování zažívá s příchodem vícejádrových procesorů na běžné uživatelské počítače velmi velký rozmach. Programátorům však správná a efektivní práce s technikami paralelního programování způsobuje značné problémy. Korektní správa a synchronizace procesů či vláken vyžaduje velké znalosti a úsilí. Ani tyto prostředky však často nezajišťují úspěch.

Tyto problémy se snaží řešit několik technologií. Jednou z nich je i technologie Grand Central Dispatch, vyvinutá společností Apple. Ta přináší několik inovativních technik, které programátorovi umožňují snadnou definici úloh a jejich paralelní zpracování. Integrací do systému také zvyšuje výkon, snižuje paměťovou náročnost aplikace a zmenšuje počet operací potřebných pro správu vláken.

Tato práce se zabývá představením knihovny Grand Central Dispatch a ukázkou jejich možností. Kapitola 3 popisuje všechny nové vlastnosti, které tato knihovna do jazyků C, C++ a Objective-C přináší. Popisuje bloky, které umožňují využít anonymní funkce známé z jiných jazyků a fronty do kterých se bloky řadí. Dále obsahuje popis skupin a zdrojů. Skupiny slouží ke sdružování bloků. Zdroje monitorují systémové objekty a jejich události.

Velmi důležitou vlastností všech paralelních nástrojů je jejich výkon. Podle autoru Grand Central Dispatch je tato knihovna velmi rychlá a ve většině případů výkonnější než ostatní způsoby. Porovnání výkonností se věnuje kapitola 3.10, která porovnává většinu volně dostupných knihoven a rozhraní pro paralelní programování.

Typickým typem aplikace, zpracovávajícím velké množství požadavků zároveň je HTTP server. Proto byl zvolen jako ukázková aplikace implementovaná jako součást této diplomové práce. Umožňuje zvolit více typů paralelního zpracování požadavků a tím změřit rozdíly jednotlivých přístupů. Server je napojen na antivirový software AVG, který je popsán v kapitole 5.

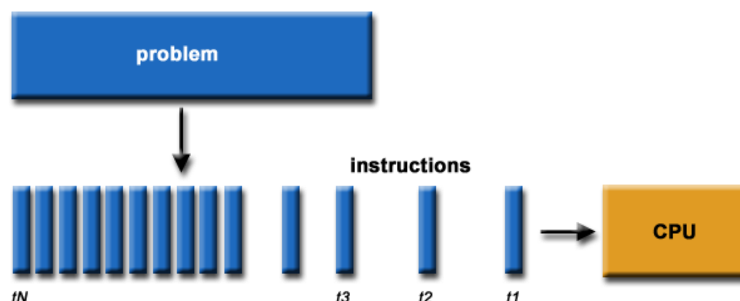
V závěrečné kapitole je knihovna zhodnocena podle dosažených výsledků, podpory a vhodnosti použití pro programátora.

Kapitola 2

Paralelní výpočty

2.1 Historie paralelního programování

Od prvopočátků byl software psán jako posloupnost příkazů, které byly zpracovávány sériově. Program byl rozdělen na diskrétní množinu instrukcí. Tyto instrukce byly zpracovávány procesorem počítače. Jakmile byla zpracována jedna instrukce, přešel procesor na další. Nemohlo být zpracováváno více instrukcí zároveň. Hlavní veličinou pro měření výkonnosti byla



Obrázek 2.1: Sériové zpracování úlohy.[7]

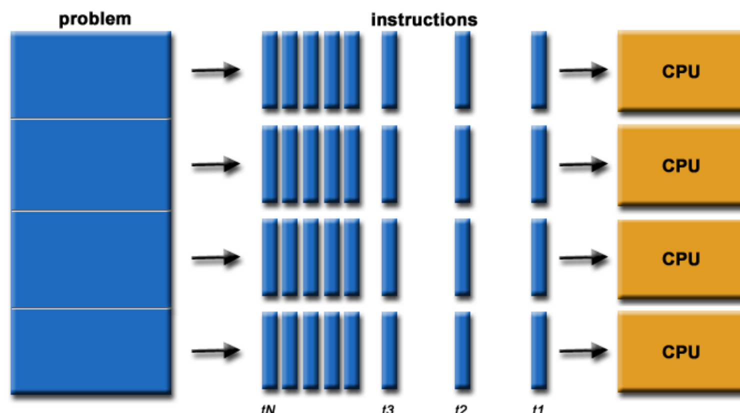
tedy frekvence procesoru počítače, s jakou dokázal zpracovávat jednotlivé instrukce. Aby bylo dosaženo zrychlení, zvyšovala se taktovací frekvence procesoru. Zpočátku se dařilo tuto frekvenci s úspěchem zvyšovat, avšak s postupem času, jak se stával návrh procesoru složitější, začali výrobci procesorů narážet na některá fyzikální omezení. Z následujícího vzorce vyplývá, že s narůstající frekvencí začala stoupat spotřeba procesoru a tím i teplo, které procesor vyzářil.

$$[\text{Spotřeba procesoru}] = [\text{Kapacita}] \times [\text{Elektrické napětí}]^2 \times [\text{Frekvence procesoru}]$$ [15]

Proto se při výrobě přešlo k novému přístupu, kterým bylo umístění více procesorových jader na jeden společný čip. Tak bylo dosaženo většího výpočetního výkonu, tedy více instrukcí za jednotku času, bez zvyšování taktovací frekvence procesoru, velikosti čipu nebo teplotní charakteristiky. [4] Tím se začaly paralelní výpočty objevovat i na běžných počítačích. V dnešní době jsou používány dvou až osmi-jádrové procesory a lze očekávat další nárůst.

Paralelní výpočet je prováděn na více výpočetních jednotkách zároveň. Úloha je rozdělena na jednotlivé části, které je možné řešit samostatně. Ty jsou dále přiděleny na jed-

notlivé výpočetní jednotky, kde jsou zpracovávány sériově. Tím se zkracuje doba potřebná k výpočtu celé úlohy. Výpočetní jednotka může být jádro vícejádrového procesoru, samostatný procesor ve víceprocesorovém stroji, samostatný počítač propojený s ostatními počítači sítí nebo kombinace předešlých možností.



Obrázek 2.2: Paralelní zpracování úlohy.[7]

2.2 Flynnova taxonomie

Flynnova taxonomie rozděluje paralelní systémy do čtyř hlavních kategorií podle toho, zda program využívá jednu nebo více sad instrukcí a zda tyto instrukce využívají jednu nebo více pamětí.

SISD Single instruction, single data odpovídá sekvenčním počítačům. Tyto počítače obsahují pouze jeden tok instrukcí, který je vykonáván sériově. Mají tedy deterministické chování. Příkladem SISD počítačů jsou pracovní stanice a notebooky s jednojádrovým procesorem.

SIMD Single instruction, multiple data jsou počítače s jedním tokem instrukcí, které mohou operovat nad více sadami dat. Tyto počítače obsahují velké množství výpočetních jednotek, které všechny provádějí stejný program. SIMD operace jsou často používány ve 3D grafice a pro zpracování obrazu a videa. SIMD procesory proto najdeme například v moderních herních konzolích.

MISD Multiple instruction, single data jsou systémy, kde více výpočetních jednotek operuje nad jednou sadou dat. Jedná se o velmi málo využívanou variantu.

MIMD Multiple instruction, multiple data je dnes nejvíce rozšířenou variantou, do které spadá většina moderních počítačů. U počítačů zařazených do této skupiny zpracovává každý procesor různý tok instrukcí s jinými daty. Takové zpracování může být jak synchronní, tak asynchronní. Také může být deterministické i nedeterministické. Tato varianta je velmi často rozšířená ve formě SPMD (single program, multiple data), kde je vykonávání jednotlivých toků instrukcí rozlišeno ve zdrojovém kódu.

2.3 Typy paralelismu

2.3.1 Aritmetický a bitový stupeň

Nejnižší stupeň paralelismu souvisí s délkou slova[1]. Zvětšení délky slova snižuje počet instrukcí procesoru potřebných k vykonání některých aritmetických operací. Tento případ nastává například při sčítání dvou 16bitových čísel na 8bitovém procesoru.

2.3.2 Instrukční stupeň

Tento typ paralelismu je závislý na překladači spouštěného programu. Ten pracuje se sledem instrukcí, které přeskládává a kombinuje do skupin. Skupiny jsou následně vykonávány paralelně tak, že jejich přeuspořádání nemá vliv na výsledný program. Instrukce v různých skupinách na sobě nemohou být datově závislé (nemohou pracovat se stejnými daty).

2.3.3 Datový paralelismus

V programových smyčkách dochází často k manipulaci s velkými datovými strukturami jako jsou např. matice. Matice je zpracovávána po řádcích či sloupcích. V takovém případě je možné jednotlivé řádky, resp. sloupce, rozdělit mezi jednotlivé výpočetní jednotky. Ty vykonávají stejnou úlohu, ale každá jednotka má přidělenu svojí část matice.

2.3.4 Funkční paralelismus

Při funkčním paralelismu vykonává každá výpočetní jednotka různé operace nad jedním nebo více soubory dat. Vykonávané operace je možné odlišit programově pomocí podmínek v kódu programu. Použití této varianty je např. při provádění několika operací nad proudem dat. Jedna výpočetní jednotka může reprezentovat např. zvukový filtr.

2.4 Modely paralelního programování

Pro paralelní zpracování úloh slouží dnes velké množství nástrojů. Některé jsou specializované nástroje velkých firem dodávajících hardware nebo software, jiné jsou open-source projekty nebo součásti operačních systémů. V této části se zaměříme na programovací techniky, které se používají na multiprocesorových počítačích. Pomineme tedy multipočítače komunikující přes síť.

2.4.1 Procesy

Proces reprezentuje samostatně prováděný program ve vlastním adresovém prostoru.[13] Skládá se z několika částí: kód programu uložený v paměti, přidělená část operační paměti, přidělené zdroje OS (popisovače souborů), informace o procesu (vlastník, rodičovský proces) a stav procesu a registrů procesoru. Tyto informace jsou uchovávány v jádře operačního systému ve struktuře zvané Process control block.

Jeden program může běžet jako více procesů. Běh více procesů je označován jako multitasking. Operační systém s pomocí plánovače za běhu procesy zastavuje, ukládá jejich

[1] Slovo je nejmenší velikost dat se kterými procesor pracuje během jednoho cyklu. V dnešní době jsou běžně používané velikosti 32 a 64 bitů. Od délky slova se odvozuje také velikost adresy dat v paměti, velikost registrů procesoru nebo velikost základních datových typů.

stav a opět je po čase spouští tak, aby měly všechny procesy spravedlivě přidělen čas na procesoru. Této změně se říká změna kontextu. Je velmi náročná, a proto byla zavedena vlákna, která budou popsána dále.

Proces může během své existence projít několika stavy. Stav vyjadřuje aktuální aktivitu procesu a je důležitý pro plánování procesů. Počet a názvy stavů se liší v jednotlivých OS. V každém lze ale nalézt tyto základní stavy:

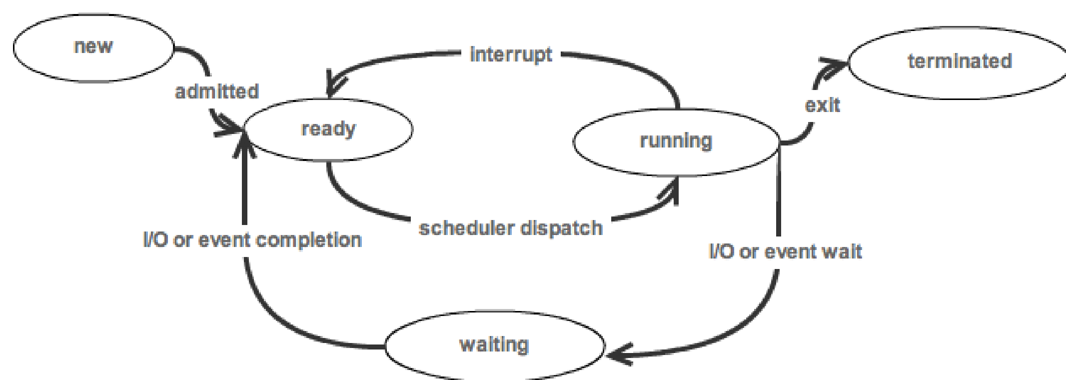
Nový (created) proces byl právě vytvořen.

Běžící (running) proces je přiřazen k procesoru a právě běží

Čekající (waiting) proces čeká na výskyt nějaké události. Např. dokončení I/O operace.

Připravený (ready) Proces čeká na přidělení k procesoru.

Ukončený (terminated) proces ukončil svůj běh.



Obrázek 2.3: Stavy procesů.[11]

Pro vzájemnou komunikaci procesu slouží nejčastěji signály. Signál je (v základní verzi) číslo (int) zaslané procesu prostřednictvím pro to zvláště definovaného rozhraní. Signály často vznikají asynchronně k činnosti programu, není tedy možné jednoznačně předpovědět, kdy daný signál bude doručen.[11]

2.4.2 Vlákna

Moderní operační systémy umožňují, aby proces obsahoval více vláken řízení. Každé vlákno je definováno svým identifikátorem, aktuální pozicí v programu, množinou registrů a zásobníkem. Pokud má proces více vláken, je schopen vykonávat více úloh současně. Vícevláknové procesy mají oproti jednovláknovému procesu několik vlastností, které jsou klíčové z hlediska jejich použitelnosti. Jsou to zejména:[11]

Responsiveness Multivláknové procesy zvyšují rychlost odezvy aplikace vůči uživateli, pokud je část aplikace blokována nebo provádí náročnou operaci. Např. webový prohlížeč umožňuje interakci s uživatelem, zatímco se v jiném vlákně nahrává obrázek.

Resource sharing Vlákna sdílejí paměť, zdroje a kód procesu. Sdílení kódu umožňuje mít více různých vláken aktivit ve stejném adresovém prostoru.

Economy Alokace paměti a zdrojů při vytváření procesu je drahá. Díky sdílení zdrojů je vytváření a přepínání kontextu vláken mnohem úspornější.

2.4.3 OpenMP

Pro snadnou práci s vlákny byla navržena knihovna OpenMP. Je dostupná pro jazyky C, C++ a Fortran. Díky tomu, že byla přijata všemi majoritními výrobci, je dostupná pro většinu moderních operačních systémů vč. Microsoft Windows. Programátorovi umožňuje ve zdrojovém kódu přidat direktivy pro preprocesor, který převede práci mezi více vláken automaticky. Tyto direktivy začínají `#pragma omp` a je jich několik typů podle určení.

Direktiva `omp parallel` slouží pro jednoduché rozdělení úlohy na více vláken. Velmi často používanou direktivou je `omp for`. Tato direktiva musí být uvedena před for cyklem, který rozdělí na více vláken podle počtu jeho iterací. Pokud je tedy P vláken a N iterací, provede každé vlákno N÷P iterací. Pro zpracování různých úloh v různých vláknech slouží direktiva `omp sections`. V ní vnořené bloky uvozené direktivou `omp section` rozdělí rovnoměrně na jednotlivá vlákna. Při rozdělení úlohy libovolnou direktivou obdrží každé vlákno svůj identifikátor, který dále slouží pro synchronizaci nebo určení části zpracovávané úlohy.

OpenMP dále obsahuje příznaky pro proměnné, které určují jejich sdílení mezi vlákny. Atribut `shared` způsobí, že proměnnou je možné číst a modifikovat ze všech vláken najednou. Atributem `private` se zajistí, že si každé vlákno vytvoří svojí lokální kopii proměnné. Je tedy soukromá pro každé vlákno. Běžně jsou jako `private` označeny počítadla iterací cyklů. Ostatní proměnné jsou standardně označeny jako `shared`. Privátní proměnné nejsou typicky inicializované. Aby byly uvnitř paralelní oblasti inicializovány, je nutné označit je jako `firstprivate`. Tím se proměnná nastaví na hodnotu před rozvětvením.

Pro synchronizaci poskytuje OpenMP několik direktiv. Pro přístup do kritické sekce slouží `omp critical`. Direktiva `omp barrier` pozdrží vykonávání programu, dokud všechna vlákna nedosáhnou této bariéry. Direktivy jako `omp for` nebo `omp parallel` ji však už v sobě obsahují, takže není nutné explicitní zadávání po každém bloku. Opakem `omp barrier` je `omp nowait`. V případě použití této direktivy jednotlivá vlákna nečekají na dokončení práce ostatních vláken, ale pokračují dále.

OpenMP obsahuje velké množství dalších direktiv, atributů a podmínek. Není však hlavním obsahem této práce, a proto tato sekce slouží pouze pro zevrubné seznámení s možnostmi této knihovny.

2.4.4 Open MPI

Knihovna Open MPI vznikla sloučením několika dosavadních implementací standardu MPI.[2] Komunikující procesy mají vlastní paměťový prostor a nesdílejí žádná data. Komunikace může probíhat dvěma způsoby. Kooperativní komunikace vyžaduje účast obou procesů na komunikaci. Data musí být jedním procesem odeslaná a druhým přijatá použitím knihovních funkcí `send` a `recv`. Jednostranná komunikace využívá vzdálené paměti. Zápis do ní poskytuje funkce `put`, čtení funkce `get`.

Open MPI je implementováno jako knihovna využívající stávajícího sekvenčního jazyka, který rozšiřuje o externí procedury pro zasílání zpráv. V dnešní době je používáno hlavně

[2] Message passing interface je standard pro programování paralelních aplikací. Procesy mezi sebou komunikují zasíláním zpráv. Mohou proto být umístěny na různých počítačích propojených sítí. MPI definuje sémantiku, syntaxi a rozhraní. Nedefinuje implementaci jednotlivých funkcí. Proto existuje několik různých implementací MPI.

v jazycích C, C++, Python, Perl nebo Java.

Na začátku jsou vytvořeny statické procesy, které se vytvoří v okamžiku spuštění paralelního programu a existují až do konce celého programu. Jejich počet je určen parametrem při spouštění programu. V průběhu programu už není možné měnit počet procesů.

Pro komunikaci mezi procesy se používají funkce `MPI_Send()` a `MPI_Recv()`. Jelikož je komunikace procesů asynchronní, je nutné zprávy odlišit. K tomu se používá délka zprávy, destinace a příznak. Tak jsou odlišeny zprávy přicházející v různých časech z různých zdrojů. Pro odlišení skupin komunikujících procesů se používají ještě tzv. komunikátory. Ty určují skupinu procesů, které mezi sebou mohou komunikovat. Nejčastější komunikátor je `MPI_COMM_WORLD`, který obsahuje všechny počáteční spuštěné procesy. Komunikace může být blokující, kdy dochází k zastavení programu a čekání na přijetí zprávy, neblokující, kdy je nutné, aby uživatel otestoval ukončení, nebo synchronní, kdy ukončení `send` vyžaduje ukončení `recv`.

Open MPI umožňuje také hromadnou komunikaci mezi procesy. Tato komunikace probíhá mezi všemi procesy v komunikátoru a je blokující a nemá vliv na ostatní komunikace jednotlivých procesů mezi sebou. Aby byla komunikace úspěšná, musí všechny procesy obsahovat volání kolektivní operace a přijímající proces musí mít správně nastavenou velikost bufferu. Implementovány jsou všechny základní druhy kolektivních operací a komunikace. Je možné proto provádět rozhlášení (broadcast), rozptyl nebo sesbírání proměnné, komunikaci všech se všemi. Nad procesy je možné provádět redukci nebo prefixovou redukci a je možné je synchronizovat bariérou.[8]

Stejně jako OpenMP, není ani Open MPI předmětem této práce a tato sekce slouží pouze jako seznámeni s možnostmi této knihovny.

Kapitola 3

Grand Central Dispatch

Grand Central Dispatch je technologie pro paralelní programování vyvinutá společností Apple a oznámená v roce 2008[9] spolu s operačním systémem Mac OS X Snow Leopard. Zveřejněna byla v roce 2009 jako open-source pod licencí Apache 2.0. Jejím hlavním účelem je zjednodušení vývoje pro stroje s více-jádrovými procesory a další SMP[1] systémy. Knihovna je dostupná i pro operační systém FreeBSD od verze 8.1. Přináší nové jazykové konstrukce, knihovny a další vylepšení systému, které byly zavedeny také do stávajících součástí systému. Přepsáno a rozšířeno bylo například CoreFoundation API, [2] API BSD subsystému [3] i Cocoa API. [4][2]

Obrázek 3.1: Logo Grand Central Dispatch.

Jedná se o knihovnu pro jazyk C, která je zakomponována hluboko do systému (viz obrázek 3.2). Nachází se v systémové knihovně `libSystem`, která dále obsahuje například knihovny pro práci s vlákny, matematickou knihovnu, standardní knihovnu jazyka C nebo knihovnu jádra operačního systému pro práci s virtuální pamětí.[10] Je možné ji využít i v jazycích C++ a Objective-C. Není tedy potřeba k aplikacím přidávat nový framework, ale stačí pouze vložit hlavičkový soubor `<dispatch/dispatch.h>`. Program je pak přeložitelný kompilátorem GCC (pouze verzí vydávanou společností Apple) nebo kompilátorem CLang. Více v kapitole 3.7.

[1]SMP je zkratka pro symetrické multiprocesorové systémy, který označuje počítače s více procesory na jedné základní desce, které sdílejí společnou paměť.

[2]API - CoreFoundation poskytuje základní datové typy a služby na úrovni pod prostředími Carbon a Cocoa.[1]

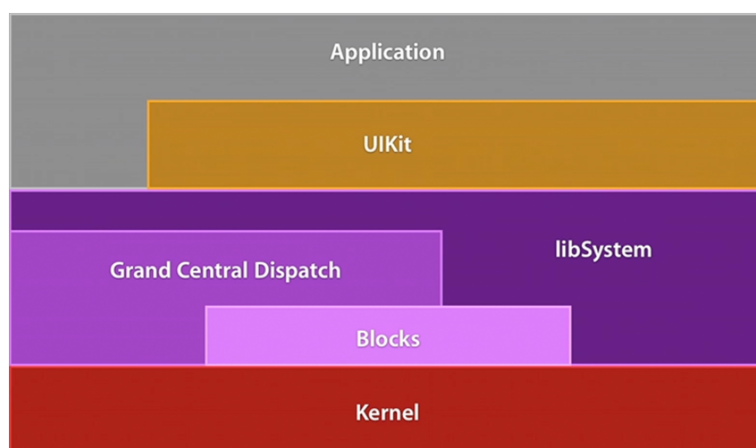
[3]Systém Mac OS X využívá některé základní systémové součásti ze systému FreeBSD. Jedná se např. o plánování úloh, podporu sítě a protokolu TCP/IP, práci s pamětí a další.[3]

[4]Hlavní framework pro tvorbu aplikací pro Mac OS X.

3.1 Úvod

Pro programátora neexistuje žádný způsob, jak zjistit kolik vláken bude moci vytvořit nebo zjistit kolik jader procesoru, na počítači, kde aplikace poběží, je k dispozici. Je proto lepší nechat rozdělení vláken a práci s nimi na jedné entitě, která je globálně dostupná v celém operačním systému. Grand Central Dispatch slouží jako tato entita, která vytváří a ruší vlákna podle toho kolik má systém procesorových jader a podle toho, jak si aplikace žádají vytvoření vláken. Pokud žádná aplikace nepožaduje vlákno, zruší GCD všechna vlákna, která dosud udržoval. Jakmile jsou vlákna požadovaná, vytvoří je zpět způsobem, který nejlépe vytěžuje dostupný hardware.

V Mac OS X jsou navíc vlákna velmi drahá záležitost. Každé z nich si udržuje vlastní sadu hodnot registrů, ukazatel na zásobník, čítač instrukcí, struktury jádra systému, které udržují prioritu plánování, nezpracované signály, masku signálů atd. To každému vláknu přidává až 512 kB dat navíc. Pokud by se vytvořilo 1000 vláken, zabraly by pouze jejich datové struktury zhruba 0,5 GB paměti. Oproti tomu je 256 B paměti navíc, které si alokuje GCD fronta, velmi zanedbatelné množství. [16]



Obrázek 3.2: Architektura systému Mac OS X. [19]

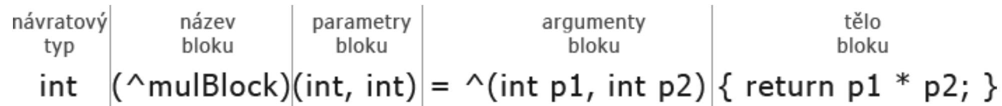
3.2 Bloky

Nejinovativnější částí GCD jsou bezesporu bloky. Tato nová jazyková konstrukce přináší do jazyků C, C++ a Objective-C nepojmenované funkce známé například z jazyka JavaScript nebo Ruby. Bloky jsou velmi podobné klasickým funkcím, ale v některých vlastnostech se liší. Reprezentuji krátké a jednoduché úlohy, které mohou být vykonávány paralelně.

```
Scheme:    (lambda (a) (add a d))
SmallTalk: 10 timesRepeat:[pen turn:d; draw]
Ruby:      z.each {|val| puts(val + d.to_s)}
C:         repeat(10, ^{ putc('0'+ d); });
```

Obrázek 3.3: Porovnání zápisu anonymních funkcí.[19]

Prvním rozdílem oproti funkcím je syntaxe. Bloky se na rozdíl od ukazatelů na funkce, které se uvozují znakem *, uvozují znakem stříška ^. Stejně jako funkce mají i bloky parametry a návratovou hodnotu. Je možné je definovat dvěma způsoby. Prvním způsobem je definice bloku jako proměnné. Blok je poté možné využít na několika místech v kódu, stejně tak jako funkci. Následující příklad ukazuje blok pojmenovaný `mulBlock`, který očekává dva parametry typu `int` a vrací hodnotu, která je také typu `int`.



Obrázek 3.4: Struktura zápisu bloku.

Pokud je blok potřeba pouze na jednom místě, je možné využít zápis přímo jako argument funkce (tzv. „inline“ zápis). V tomto případě se blok zapisuje následovně:

```
void printInt(int (^block)(void)) {
    printf("%d\n", block());
}

int pivot = 3;
for(int i=0; i < 10; i++)
    printInt(^{ return i*pivot; });
```

Kód 3.1: Použití inline zápisu bloku.

Další vlastností bloků, kterou se liší od funkcí, je zachytávání lexikálního rozsahu prostředí, ve kterém je blok definován. Tato vlastnost je v jiných programovacích jazycích známa jako uzávěr (closure). Uvnitř bloku je tedy možné přistupovat k proměnným, které byly definovány před místem vytvoření bloku. Vytvořením bloku je myšlena jeho definice ve zdrojovém souboru. Není tak možné upravovat lokální proměnné z pojmenovaných bloků, které byly definovány na globální úrovni. Sdílené proměnné jsou automaticky nakopírovány na zásobník. Uvnitř bloku je pak možné je číst i upravovat. Změny těchto proměnných se však nepromítnou mimo blok, jelikož jsou to pouze lokální kopie.

V případě, že je potřeba, aby se změny proměnné projevily i mimo blok, je nutné přidat před deklaraci proměnné modifikátor `__block`. Proměnná je poté udržována ve společném kontextu bloku i prostředí, ze kterého je blok vytvořen a její změna se projeví na obou místech. Tento modifikátor lze však přidat pouze lokálním proměnným.

```
__block int mul = 1;
typedef int (^Multiplicator)(int);
Multiplicator mulBlock = ^(int base){
    int res = base * mul++;
    return res;
};

printf("%d\n", mulBlock(10));    // 10
printf("%d\n", mulBlock(10));    // 20
```

Kód 3.2: Modifikace proměnné mul uvnitř bloku, která se promítne i mimo blok.

Bloky je možné předávat funkcím stejně jako jakýkoliv jiný datový typ. Ve většině případů je není nutné deklarovat, ale stačí je implementovat přímo ve volání funkce. Standardem je, že funkci se předává pouze jeden parametr a vždy na posledním místě v seznamu parametrů. Typickým příkladem funkcí s blokem jako parametrem jsou všechny funkce knihovny libdispatch.

```
dispatch_apply(count, queue, ^(size_t i) {
    printf("%u\n", i);
});
```

Kód 3.3: Předání bloku jako parametru funkci.

3.3 Fronty

Fronty jsou mechanismus, který slouží ke zpracování jednotlivých úkolů. Jsou to struktury podobné objektům spravované operačním systémem, které si udržují seznam vložených úloh.[4] Umožňují vykonávání úloh sériově nebo paralelně. Lze pomocí nich provádět jakékoliv operace, které je možné provádět s vlákny. Výhodou front oproti vláknům je však jejich jednoduché použití a větší výkon oproti obdobnému kódu využívajícímu vlákna. Práce s vlákny, jako je jejich vytváření, ukončování a zjišťování kolik vláken je možné vytvořit, je plně v režii fronty.

Úlohy mohou reprezentovat například složité výpočty, přístup k blokovým zařízením nebo zápis do souboru. Takové operace mohou zablokovat chod programu nebo uživatelského rozhraní, a je proto vhodné zpracovávat je odděleně. Do front jsou předávány ve formě funkcí nebo bloků. Je nutné tedy tyto úlohy navrhovat tak, aby mohly běžet nezávisle a samostatně.

Všechny fronty jsou typu FIFO. Úlohy v jedné frontě jsou tedy zpracovávány v takovém pořadí v jakém jsou vkládány do front. Dvě úlohy ve dvou frontách mohou však běžet paralelně. Počet úloh, které běží zároveň určuje dynamicky operační systém. Není tedy zaručeno, že v případě, že je vytvořeno 100 front, poběží 100 úloh.

Existují 3 typy front, které se liší způsobem použití a přístupem k nim.

3.3.1 Sériové fronty

Sériové fronty, označované také jako privátní, zpracovávají úlohy postupně jednu za druhou tak, že v daný okamžik běží pouze jedna úloha z fronty. Úloha je spuštěna v samostatném vlákně, které se pro každou úlohu může lišit.

Na rozdíl od globálních paralelních front, které jsou vytvářeny automaticky, je nutné tyto fronty explicitně vytvářet a udržovat. Front tohoto typu je možné vytvořit libovolné množství. Tyto fronty pracují paralelně a nezávisle na sobě. Pokud jsou vytvořeny např. 4 sériové fronty, zpracovává fronta pouze jednu úlohu najednou, ale v jeden okamžik běží celkem 4 úlohy. U každé fronty je zadán také její název, který pomáhá při práci s ladícími nástroji.

```
dispatch_queue_t fronta;
fronta = dispatch_queue_create("nazev.fronty", NULL);
```

Kód 3.4: Příklad vytvoření fronty.

Stejně jako všechny ostatní objekty knihovny, jsou i fronty objekty s počítadlem referencí. Při vytvoření fronty se nastaví počáteční hodnota počítadla na 1. Pro manipulaci s touto hodnotou slouží funkce `dispatch_retain` zvyšující počítadlo o 1 a `dispatch_release`, která počítadlo snižuje o 1. Jakmile počítadlo klesne na 0, je fronta systémem automaticky uvolněna. Pokud je potřeba využívat frontu mělo by se zvýšit počítadlo a po vykonání všech operací, kdy už není fronta potřeba, by se mělo opět snížit.

3.3.2 Paralelní fronty

Fronta slouží pro paralelní zpracování několika úloh najednou, avšak stále v pořadí, v jakém byly do fronty vloženy. Úlohy běží v oddělených vláknech, která jsou dynamicky vytvářena a přidělována frontou podle systémových podmínek. Systém tak rozhoduje, kolik může být najednou zpracováno úloh. Typicky to určí podle počtu dostupných procesorů. Paralelní fronty nelze vytvořit v programu, ale jsou vytvářeny globálně. Pracuje se s nimi pouze pomocí jednoduchého objektu typu `dispatch_queue_t`. Tento objekt vrací funkce `dispatch_get_global_queue`. Je možné získat celkem 3 různé fronty různých priorit podle parametru funkce pro získání fronty. Výčet `dispatch_queue_priority_t` obsahuje tyto symbolické konstanty:

DISPATCH_QUEUE_PRIORITY_HIGH Fronta s úlohami se zvýšenou prioritou. Úlohy z této fronty jsou vždy provedeny jako první.

DISPATCH_QUEUE_PRIORITY_DEFAULT Standardní priorita úloh. Úlohy z fronty jsou zpracovány po provedení úloh se zvýšenou prioritou, ale před provedením úloh s nízkou prioritou.

DISPATCH_QUEUE_PRIORITY_LOW Fronta obsahující úlohy s nízkou prioritou. Tyto úlohy jsou zpracovány až po zpracování úloh z obou předcházejících front.

```
dispatch_queue_t fronta;
fronta = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, NULL);
```

Kód 3.5: Příklad získání odkazu na globální konkurentní frontu.

I když jsou tyto fronty objekty, u kterých se počítá počet referencí, není nutné je po skončení práce explicitně uvolňovat. Všechna uvolnění jsou ignorována. Proto není potřeba ani udržovat referenci na frontu, ale vždy stačí zavolat funkci `dispatch_get_global_queue`.

3.3.3 Hlavní fronta

Hlavní fronta je sériová fronta dostupná globálně z celého programu, která vykonává úlohy v hlavním vlákně aplikace. Tato fronta spolupracuje s hlavní smyčkou programu a prokládá její běh úlohami z hlavní fronty. Jelikož běží v hlavním vlákně, je často používána jako synchronizační bod v rámci aplikace.

```
dispatch_async(q, ^{
    NSArray* uzivatele = nactiUzivatele();

    dispatch_async(dispatch_get_main_queue(), ^{
        zobrazUzivatele(uzivatele);
    });
});
```

Kód 3.6: Práce s hlavní frountou.

Předchozí zdrojový kód ukazuje využití hlavní fronty. Seznam uživatelů je načten paralelně s během hlavního vlákna a po načtení výsledků je v hlavním vlákně zavolána funkce na zobrazení seznamu uživatelů. Operace načítání, která může stahovat data ze sítě a může být velmi pomalá neblokuje hlavní program a uživatelské rozhraní. Funkce vykreslení, která je rychlá a změní část uživatelského rozhraní je volána z hlavního vlákna.

3.3.4 Operace s frontou

Základní operace, která je s frontou prováděna je zařazení nového bloku na její konec. To je možné provést několika způsoby. Synchronní zařazení bloku provádí funkce `dispatch_sync`. Při tomto zařazení se čeká na výsledek bloku. Program je tedy blokován. Tuto varianta se nedoporučuje využívat, jelikož není možné zjistit, kdy se daný blok provede. Je proto lepší využívat asynchronní variantu - funkci `dispatch_async`, která po zařazení bloku do fronty nečeká na provedení bloku a pokračuje dál ve vykonávání. Tuto funkci je vhodné použít zejména pokud se pracuje v hlavním vlákně aplikace, kdy by při použití synchronní varianty došlo např. k zablokování uživatelského rozhraní. Přesto existují případy, kdy je vhodnější využít synchronní variantu. Hodí se například na místa, kde by se jinak musela řešit synchronizace bloků při přístupu k některým proměnným atp.

Obě funkce mají ekvivalenty s příponou `_f`, které místo bloků přidávají do fronty úlohy reprezentované funkcemi. Další variantou těchto dvou funkcí jsou funkce přidávající skupiny úloh (popsané v kapitole 3.4) do fronty. Tyto funkce mají tvar `dispatch_group_sync` a `dispatch_group_async` a fungují zcela stejným způsobem, pouze obsahují navíc parametr specifikující skupinu do které úlohy patří.

```
dispatch_queue_t q;
q = dispatch_queue_create("cz.vutbr.fit.xsvecm07", NULL);

dispatch_async(q, ^{ printf("Asynchronne zarazeny blok."); });
```

Kód 3.7: Asynchronní zařazení bloky do fronty.

Při klasickém programování jsou hojně využívány takzvané callback funkce. To jsou funkce, které se vykonají po dokončení jiné funkce. Knihovna GCD na to nemá žádný zvláštní mechanismus, jelikož se opět jedná o jednoduchý kus kódu, který se zařadí do fronty na konci jiného bloku. Příklad simulace callback funkce ukazuje následující zdrojový kód počítající průměrnou hodnotu prvků pole, kde je využita nová funkce pro simulaci `dispatch_async` s blokem, který se po dokončení přidá do zadané fronty. Callback blok a fronta ve které se provede jsou předány jako parametry funkce.

```
void average_async(int *data, size_t len,
                   dispatch_queue_t queue, void (^block)(int)) {
  dispatch_retain(queue);

  dispatch_async(
        dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0),
        ^{
           int avg = average(data, len);
           dispatch_async(queue, ^{ block(avg);});

           dispatch_release(queue);
        });
}
```

Kód 3.8: Simulace callback funkce.

Stejně jako zdroje je fronty možné pozastavit a znovu obnovit. Volání `dispatch_suspend` frontu pozastaví a zvýší počítadlo pozastavení (obdobné jako počítadlo referencí). Pro obnovení činnosti nebo snížení počítadla slouží funkce `dispatch_resume`. Dokud je počítadlo větší než nula, tak je fronta pozastavená. Je proto nutné vždy vyvážit počet pozastavení a počet obnovení. Uspávání a obnovení je asynchronní operace a proto neovlivní průběh aktuálně zpracovávaného bloku. Fronta se vždy zastaví až po jeho dokončení.

3.3.5 Paralelní zpracování cyklu

Grand Central Dispatch nabízí podobnou možnost zpracování cyklu jako je například i v OpenMP. Pokud je v kódu například `for` cyklus a jednotlivé jeho iterace jsou na sobě nezávislé a nezáleží v jakém pořadí se dokončí, je možné tento cyklus nahradit voláním funkce `dispatch_apply` nebo `dispatch_apply_f`. To rozdělí jednotlivé úlohy do fronty. Pokud se jedná o paralelní frontu je dosaženo zpracovávání více úloh najednou. Předání úloh do sériové fronty je také možné, ale nemá žádné výhody oproti původní implementaci cyklu.

U krátkých rychlých úloh se projevuje režie vytváření potřebných struktur a tak nemusí být efektivita `dispatch_apply` tak velká. Pro vylepšení se používá trik, kdy blok nereprezentuje pouze jeden, ale několik několik výpočtů najednou. Pomocí stanovení kroku a vložení vnitřního cyklu tak každá úloha ve frontě počítá více iterací a zlepšuje se poměr režie a doby provádění jedné úlohy. Následující příklad ukazuje rozdělení cyklu na několik úseků pomocí kroku a využití funkce `dispatch_apply` k paralelizaci jejich zpracování:

```
int krok = 20;
dispatch_queue_t q =
        dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

dispatch_apply(pocet / krok, q, ^(size_t i){
    size_t j = i * krok;
    size_t j_stop = j + krok;
    do {
        printf("soucasny index: %u\n", (unsigned int) j++);
    }while (j < j_stop);
});

size_t i;
for (i = pocet - (pocet % krok); i < pocet; i++)
    printf("%u\n", (unsigned int)i);
```

Kód 3.9: Paralelizace cyklu s volitelným krokem.

Protože po dělení počtu iterací krokem zbývá zbytek, který se do fronty nezařadí, proběhne jeho zpracování v hlavním vlákně.

Pro optimální zvolení velikosti kroku je nutné testovat více hodnot, jelikož určení není jednoznačnou záležitostí a závisí na náročností výpočtu každé iterace a počtu dostupných protředků.

3.4 Skupiny

Pokud je zpracováváno více různých úloh, které mohou být umístěny i do jiných front, může být potřeba provést akci po dokončení všech těchto úloh. K tomuto účelu slouží v GCD skupiny. Pomocí nich je možné jak synchronně, tak asynchronně monitorovat úlohy. Následující kód ukazuje jak zařadit skupinu úloh do fronty a počkat na dokončení všech úloh v dané skupině.

```
dispatch_group_t group = dispatch_group_create();
dispatch_queue_t queue =
    dispatch_get_global_qeueue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
for(id obj in array)
    dispatch_group_async(group, queue, ^{
        [self doSomethingIntensiveWith:obj];
    });
```

```
dispatch_group_wait(group, DISPATCH_TIME_FOREVER);
dispatch_release(group);
```

Kód 3.10: Zařazení skupiny úloh do fronty (použito Objective-C).

Tento způsob provedení je blokující, jelikož se čeká na provedení všech úloh ve frontě i přesto, že jsou zařazeny do paralelní fronty. Program přestane odpovídat dokud se všechny úlohy nezpracují. Je možné vypustit volání `dispatch_group_wait` a nahradit ho funkcí `dispatch_group_notify`, která po dokončení všech úloh ve skupině zajistí provedení bloku, který má jako třetí parametr.

```
dispatch_group_notify(group, queue, ^{
    printf("Group ended.");
});
```

Kód 3.11: Asynchronní callback po dokončení všech úloh ve skupině.

3.5 Semafory

Semafory v GCD jsou velice podobné klasickým semaforům. Vyžívají se v případech kdy se do fronty přidávají operace se zdrojem, který má omezený přístup. Oproti systémovým semaforům se liší v efektivitě, kde dosahují lepších výsledků.[4] Tyto semafory provádí volání jádra pouze v případě, kdy musí být volající vlákno zablokováno z důvodu nedostupnosti zdroje. Pokud je zdroj dostupný, tak nedochází k volání jádra.

Semafory se vytvářejí funkcí `dispatch_semaphore_create` ve které je celým kladným číslem specifikován počet dostupných zdrojů. Ve výpočtu stačí pouze zavolat funkci `dispatch_semaphore_wait`, která zablokuje vykonávání programu do doby, než je zdroj přístupný. Doba čekání na zdroj je omezena druhým parametrem této funkce, který určuje maximální možný čas čekání. Po dokončení práce a uvolnění zdroje je nutné signalizovat opuštění zdroje funkcí `dispatch_semaphore_signal`. Následující zdrojový kód ukazuje využití semaforu k omezení přístupu k souboru pouze pro 10 vláken.[4]

```
dispatch_semaphore_t fd_sema = dispatch_semaphore_create(10);

dispatch_semaphore_wait(fd_sema, DISPATCH_TIME_FOREVER);
fd = open("/etc/services", O_RDONLY);

close(fd);
dispatch_semaphore_signal(fd_sema);
```

Kód 3.12: Ukázka využití semaforu pro přístup k souboru.

3.6 Zdroje

Při práci se systémovými komponenty je vždy nutné počítat s velkou režií jednotlivých úloh. Volání jádra nebo práce se soubory vyžaduje přepnutí kontextu a to je, v porovnání s voláním uvnitř procesu, velmi drahá záležitost. Velmi mnoho systémových knihoven nabízí asynchronní rozhraní pro přístup k systémovým prostředkům, které využívají tzv. callback funkce, které jsou provedeny po vykonání požadavku.

Zdroje (Dispatch Sources) slouží k usnadnění obsluhy specifických nízko-úrovňových událostí. Eliminují tak callback funkce, které jsou předávány jako parametry při obsluze událostí. V případě, že systém zaznamená požadovanou událost, zařadí odpovídající blok

do fronty. Až do doby jejich explicitního zrušení slouží zdroje jako automatický nepřetržitý zdroj událostí. Události mohou nastávat v nepravidelných intervalech, proto si zdroje udržují vlastní frontu pro zařazování úloh a zabraňují tak předčasnému zrušení fronty v případech, kdy ještě existují úlohy čekající na zpracování.[4]

Grand Central Dispatch podporuje tyto typy zdrojů

Časovač periodicky generuje události

Signál upozorňuje na UNIXové signály

Soubor změny v souboru (a také socketu) jako například:

- soubor je připraven pro zápis
- soubor je připraven pro čtení
- soubor byl smazán, přesunut nebo přejmenován
- změnily se meta-informace o souboru

Proces upozorňuje na události procesu jako jsou:

- proces skončil
- proces zaznamenal volání fork nebo exec
- proces obdržel signál

Mach jádro události generované jádrem systému[5]

Vlastní uživatelem definované a vyvolávané události

3.6.1 Vytvoření zdroje

Jelikož vytvoření fronty vyžaduje další dodatečná nastavení, je po vytvoření zdroj suspendován. V tomto suspendovaném stavu může zdroj přijímat úkoly do fronty, ale nezpracovává je. Vytvoření zdroje zajišťuje funkce `dispatch_source_create`. Ta potřebuje typ zdroje, jeho identifikátor, masku specifikující které události sledovat a identifikátor fronty do kterého jsou úlohy řazeny.

Postup vytvoření zdroje je tedy následující:

- Vytvořeni zdroje funkcí `dispatch_source_create`
- Nastavení zdroje
  - Nastavení obsluhy události
  - Nastavení časovače u událostí generovaných pravidelně
- Nastavení obsluhy ukončení generování událostí
- Zavolání funkce `dispatch_resume` pro zahájení zpracování událostí

[5] Mach je jádro operačního systému vyvinuté na Carnegie Mellon University. Jedná se o mikrojádro z něhož vychází jádra např. Mac OS X nebo GNU Hurd.

3.6.2 Nastavení obsluhy zdroje

Po vytvoření zdroje je nutné nastavit funkci nebo blokový objekt, který bude zajišťovat obsluhu a zpracování sledované události. Možnosti jak nastavit obsluhu jsou dvě. Funkce `dispatch_source_set_event_handler` a `dispatch_source_set_event_handler_f` nastaví blokový objekt, resp. funkci, která po vyvolání události zpracuje.

Pokud je událost už zařazena ve frontě a dorazí nová událost, sloučí se tyto dva výskyty do jednoho. Obslužná funkce nebo blok bere tuto událost jako jednu, avšak podle typu události je možné zjistit informace i o druhé události, která byla sloučena. V jednu chvíli je zpracovávána pouze jedna událost. Pokud se tedy vyskytne nová událost když je ještě zpracovávána předchozí událost, je nová událost zařazena do fronty a zpracována až v případě, že je fronta volná.

Obsluha založená na funkci přebírá jako parametr ukazatel na paměť obsahující kontext. Obslužný blok nemá žádný parametr, jelikož zachytává kontext standardně. Ani jedna z obslužných metod nemá návratovou hodnotu.

```
// obsluha události blokem
void (^dispatch_block_t)(void)

// obsluha události funkcí
void (*dispatch_function_t)(void *)
```

Informace o obsluhované události uvnitř obslužné funkce nebo bloku je možné získat přímo z události. K získání těchto informací slouží metody:

dispatch_source_get_handle Tato funkce vrací identifikátor datového typu, který zdroj sleduje.

- číslo typu `int` pro zachycený signál nebo změněný soubor
- strukturu `pid_t`, která reprezentuje sledovaný proces
- strukturu `mach_port_t` pro událost Mach jádra
- pro ostatní zdroje je návratová hodnota nedefinovaná

dispatch_source_get_data Funkce vrací jakákoliv data asociovaná s událostí.

- počet bytů dostupných pro čtení v případě, že se jedná o událost čtení ze souboru nebo socketu
- kladný integer, pokud je možné do souboru nebo socketu zapisovat
- konstantu definovanou v `dispatch_source_vnode_flags_t` pokud je sledována změna souboru v souborovém systému
- konstantu z `dispatch_source_proc_flags_t` v případě sledování události procesu
- jednu z konstant ve výčtovém typu `dispatch_source_machport_flags_t` pro události Mach jádra vrací
- pro vlastní zdroje vrací novou datovou hodnotu vytvořenou z existujících dat a nových dat předaných funkci `dispatch_source_merge_data`

dispatch_source_get_mask Funkce pro zjištění masky, která byla použita při vytváření zdroje.

3.6.3 Zrušení zdroje

Pokud není zdroj zrušen, zpracovává události až do konce běhu programu. Je však možné ho explicitně zrušit funkcí `dispatch_source_cancel`. Ta zastaví zachytávání nových událostí, které nemůže být opětovně spuštěno. Tato operace je asynchronní. Požadavky už zařazené ve frontě jsou zpracovány a na konci je zavolán tzv. `cancellation handler` (pokud je nastaven).

`cancellation handler` slouží pro vykonání operací, které jsou potřeba provést ještě před uvolněním zdroje z paměti. Takovou operací může být uvolnění systémových prostředků alokovaných zdrojem, uzavření souboru nebo socketu atp. Nastavení handleru je volitelné a provádí se funkcí `dispatch_source_set_cancel_handler`, která bere jako parametr sledovaný zdroj a blok s obsluhou nebo `dispatch_source_set_cancel_handler_f`, která přebírá zdroj a funkci s obsluhou zrušení zdroje.
Příklad jednoduché obsluhy ukončení čtení ze souboru:

```
dispatch_source_set_cancel_handler(zdroj, ^{
   close(soubor);  // soubor je popisovac souboru zachyceny z kontextu
});
```

3.6.4 Další operace se zdroji

V případě potřeby je možné zdroji změnit frontu, do které řadí na zpracování jednotlivé úlohy. Tato vlastnost se hodí například pokud je nutné změnit prioritu úkolů. Změna se provádí funkcí `dispatch_set_target_queue` a jedná se o synchronní operaci, která je provedena co nejrychleji. Pokud je však nějaký požadavek zařazen ve frontě, je v ní také vykonán a nová fronta je použita pro všechny následující požadavky.

Zdroje je také možné funkcí `dispatch_suspend` uspat a funkcí `dispatch_resume` opět probouzet k činnosti. Tyto funkce zvyšují a snižují čítač uspání. Pozastavení musí být proto udržováno v rovnováze s probuzením. V případě častějšího uspání než probuzení zůstává zdroj uspán. Pokud je zdroj zdroj uspaný a je zachycena sledovaná událost, je událost zaznamenána. Po probuzení zdroje nejsou všechny události doručeny za sebou, ale jsou sloučeny a doručena je pouze poslední z nich. Toto chování zabraňuje zahlcení fronty po probuzení zdroje. Příkladem je vícenásobné přejmenování souboru po dobu uspaného zdroje, který monitoruje přejmenování. Po jeho probuzení je mu doručena pouze událost s posledním názvem, na který byl soubor přejmenován. [4]

3.7 Podpora GCD v operačních systémech

Knihovna Grand Central Dispatch se poprvé objevila v operačním systému Mac OS X 10.6. Apple ji uvolnil jako otevřený software, čehož se krátce po uvolnění chopil tým lidí pracujících na systému FreeBSD a za několik dní technologii portoval i pro tento operační systém. Standardní součástí se GCD stalo od verze 8.1. Od této verze výše funguje GCD bez problémů. V případě, že nejsou potřeba bloky, je možné překládat kompilátorem GCC, který neobsahuje pro bloky podporu. Pro možnost použití bloků je nutné využívat kompilátor Clang, který však nepodporuje C++. Programátor je tedy v jednom zdrojovém souboru používat buď C++ nebo bloky, ale ne obojí dohromady.

Přenos na jiné platformy je velmi komplikovaný, jelikož vyžaduje integraci přímo do kernelu operačního systému. Vzhledem k tomu, že Mac OS X používá hybridní jádro vycházející z jádra Mach a z jádra BSD, vyžadoval přenos GCD pod BSD přizpůsobení kon-

venčnějšímu BSD jádru bez vrstvy Mach a použití POSIX semaforů místo semaforů Mach jádra. Současně s úpravou kernelu je nutná také úprava kompilátoru. Díky kompilátoru Clang a virtuálním stroji LLVM[6] je toho možné dosáhnout s minimálním úsilím. Programy je také možné překládat kompilátorem GCC. Podporu pro GCD však obsahuje pouze verze vyvíjená společností Apple.[20]

Počátkem roku 2011 byla vyvinuta také verze GCD pro operační systémy Linux a Windows. Autorem je Marius Zwicker, který knihovnu uvolnil pod názvem libXDispatch. Tato knihovna poskytuje podporu pro GCD i v systémech Winodows XP, Windows 7, Debian 6.0, openSUSE a Ubuntu. Na těchto systémech byla úspěšně otestována. Očekává se však, že bude bez problémů fungovat i v dalších systémech.[14]

3.8 Přepis existujících aplikaci do GCD

Vzhledem k tomu, že GCD je nová knihovna přinášející zcela nový styl paralelního programování, není možné jednoduše transformovat stávající programy využívající vlákna nebo NSOperation[7] objekty do formy podporující GCD. Přepsáním programu se zredukují prostředky, které aplikace vynakládá na správu a uložení zásobníků vláken, ukládaných v paměťovém prostoru aplikace, eliminuje se kód potřebný pro vytváření vláken a distribuce práce mezi ně a celkový kód aplikace je přehlednější a jednodušší.

Nejčastější změnou bývá přepsání kódu s vlákny na kód využívající fronty. Obvykle vlákna zpracovávají jednu úlohu. Tato vlákna je možné nahradit jednou paralelní frontou do které jsou přidávány úlohy stejně jako při vytvoření vlákna a jeho spuštění. Pro vlákna do kterých jsou řazeny úlohy na zpracování postupně je nutné zvolit zpracování úloh. Pokud se jedná o sériové zpracování nebo synchronizované spouštění, hodí se jako náhrada sériová fronta. Pokud nezáleží na pořadí a úlohy mohou být zpracovány nezávisle na sobě, používá se globální paralelní fronta. Pro transformaci návrhového vzoru `Thread Pool` se využívá také globální paralelní fronta do které jsou řazeny jednotlivé požadavky.

Prosté přepsání nemusí být vždy to pravé a může způsobit problémy. Zejména při přístupu ke sdíleným zdrojům může nastat nedefinované chování. Nejlepším opatřením je co největší eliminace společných částí. Pokud nelze závislosti odstranit, je nejlepší volbou využití sériové fronty. Stejně tak se hodí pro nahrazení kódu využívajícího zámky k synchronizaci přístupu do kritické sekce. Odpadají tak problémy např. se soutěžením procesů o přístup do kritické sekce, jejich vzájemné synchronizace atd.

Pro nahrazení spojení vláken v určitém okamžiku vykonávání programu slouží v knihovně Grand Central Dispatch skupiny úkolů. Stejně jako spojení vláken, jsou skupiny mechanismem jak zastavit provádění vlákna do doby než se vykonají všechna vytvořená dceřiná vlákna.

Pokud jsou vlákna využívána pro čtení nebo zápis do souboru či monitorování operací se souborem, je možné je nahradit zdroji událostí popsanými v kapitole 3.6.

3.9 Praktické využití GCD

I přesto, že se jedná o poměrně mladou technologii, je již GCD využíváno. Nejznámější implementace, která není součástí Mac OS X je modul pro paralelní zpracování ve webovém server Apache. Vývojář Robert Watson chtěl pomocí tohoto modulu ukázat způsob využití

[6] Low Level Virtual Machine

[7] NSOperation jsou objekty Cocoa frameworku zapouzdřující úlohy tak, jako bloky v knihovně GCD.

GCD, výhody plynoucí z jeho využití a jednoduchost integrace do již zavedených projektů. Celá implementace má pouhých 950 řádků, oproti původní vláknové implementaci, která měla 1,6-3x více řádků. [21]

3.10 Výkonnost

GCD je podle slov společnosti Apple výkonnější než ruční implementace vláken. Protože jsou vlákna spravována operačním systémem, který vytváří pouze takový počet vláken, která jsou potřeba, měl by teoretický výkon být vyšší, než při správě vláken programem, který navíc neví, kolik má k dispozici jader a systémových prostředků a vlákna vytváří podle potřeby algoritmu. Další nevýhodou vláken je při jednoduché implementaci nemožnost dále škálovat výkon. V případě knihovny GCD zajišťuje škálování sám operační systém tak, že navýší počet vláken, která zpracovávají úlohy ve frontách. Tím se urychlí celková doba zpracování všech úloh. Toto škálování je automatické a programátor se nemusí starat o zjišťování počtu jader procesoru. Aplikaci tak stačí přenést na počítač s více jádry a aplikace se sama přizpůsobí. V případě využití vláken a návrhového vzoru „Thread pool“ se toto chování do jisté míry simuluje, ale stále nedochází k tak efektivnímu využití dostupných prostředků.

3.10.1 Způsob měření

Pro ověření a porovnání výkonnosti s ostatními způsoby paralelizace úloh byl použit program `Dispatch_Compared` vydaný přímo společností Apple, který je dostupný na stránkách `developer.apple.com` pro vývojáře. Program měří dobu provádění dvěma způsoby.

Prvním způsobem je doba několikanásobného výpočtu složité trigonometrické funkce pomocí různých způsobů paralelizace výpočtu, které jsou popsány v následujícím seznamu (názvy metod jsou důležité kvůli orientaci ve výsledcích měření popsaných dále v dokumentu):

loop Zpracování sériově ve smyčce. Tato varianta neobsahuje žádnou paralelizaci a má nulový overhead.

apply Paralelizace funkcí `dispatch_apply`, která rozdělí cyklus na části a každou část zpracuje paralelně s ostatními částmi.

serial Vytvoření sériové fronty, asynchronní zařazení úloh v blocích do fronty a vyčkání na provedení všech úloh.

parallel Asynchronní zařazení úloh do globální paralelní fronty se standardní prioritou a vyčkání na vyprázdnění fronty.

queues Vytvoření samostatné sériové fronty pro každou úlohu. Sériové fronty zpracovávají úlohy nezávisle na sobě, takže dochází k paralelizaci jednotlivých úloh. Funkce končí po dokončení všech front.

openmp Využití OpenMP pro paralelizaci `for` cyklu, který zpracovává jednotlivé úlohy. Iterace cyklu jsou rozděleny na části, které jsou následně zpracovány paralelně.

thread Pro každou úlohu je vytvořeno samostatné vlákno ve kterém je úloha zpracována. Po skončení všech vláken (`pthread_join`) je dokončen výpočet.

Počítaná trignometrická rovnice je tvaru:

$$x = \tan \pi - \arctan e^{2*\log \sqrt{x}}$$

Výpočet této rovnice je prováděn v několika iteracích, kdy se neustále mění proměnná `x`. Počet iterací výpočtu `x` je možné nastavit jako argument programu: `-f počet`. Po provedení všech iterací se zaznamená jejich celkový čas, druhá mocnina celkového času, systémový[8] a uživatelský[9] čas. Tyto 4 hodnoty se přičtou k hodnotám z předchozího výpočtu a začne se počítat znovu.

Výpočty se provádějí po předem přidělený čas. Standardně se jedná o 60 vteřin. Změnu je možné provést použitím argumentu programu: `-t čas`. K zastavení tohoto cyklu je velmi dobře využito funkce knihovny Grand Central Dispatch. Doba testování je totiž omezena proměnnou `test_running`, která je při spuštění nastavena na hodnotu 1. Ještě před tímto nastavením je do globální fronty s vysokou prioritou zařazen funkci `dispatch_after` blok obsahující nastavení proměnné `test_running`. `Dispatch_after` je funkce, která jako první parametr bere čas, za který se má vložený blok provést.

Po skončení časového bloku, který je výpočtu přidělen je součet všech časů vydělen počtem iterací a tím jsou zjištěny průměrné časy jednoho výpočtu. Na standardní výstup se dále tiskne chyba měření, zrychlení a režie spojená s výpočtem. Tyto hodnoty jsou vypočítány podle následujících vzorců:

$$error = \sqrt{wall_sq - t_wall^2}$$

$$speedup = \sqrt{\frac{b_wall}{t_wall}}$$

$$overhead = \frac{t_user - t_system}{b_user - b_system}$$

Proměnné s předponou `t_` se vztahují k jednomu výpočtu, proměnné s předponou `b_` označují první výpočet se kterým se všechny další výpočty porovnávají.

error je chyba měření,

wall_sq je průměrná druhá mocnina doby trvání výpočtu,

wall je průměrná doba trvání aktuálního měřeného výpočtu,

system čas, který procesor stráví v privilegovaném režimu,

user čas procesoru v uživatelském režimu

Druhým způsobem měření bylo pouhé vyvolání paralelního kódu bez provádění nějakého výpočtu. Měřila se tak efektivita volání jednotlivých implementací paralelního zpracování, bez režie související s výpočtem. Obsahem těchto funkcí je pouze přiřazení do proměnné, nebo vrácení NULL. Testování probíhalo na následujících metodách:

alloc Alokace pole o `n` položkách.

[8] Čas procesoru strávený v uživatelském režimu. Procesor v tomto režimu pracuje s daty aplikace.

[9] Čas, který procesor strávil v systémovém režimu. V tomto režimu může procesor provádět jakékoliv operace.

array Přidání položek do datového typu `MutableArray`, což je datový typ reprezentující pole s proměnlivou velikostí. V C++ STL je možné ho přirovnat k vektoru (datový typ `vector`).

dsptch_f Vytvoření sériové fronty a zařazení jednoduchých úloh reprezentovaných funkcí.

dispatch Vytvoření fronty a zařazení jednoduchých úloh reprezentovaných bloky.

fork I přes zavádějící název se jedná o vytvoření `n` vláken s přiřazenou funkcí vracející NULL.

3.10.2 Výsledky synchronního měření

Měření výkonnosti bylo prováděno na počítači MacBook White s 2,13 GHz Intel Core 2 Duo procesorem (2 jádra procesoru, 1,07GHz sběrnice), 4GB 800 MHz DDR2 SDRAM operační pamětí a diskem o rychlosti 7200 ot./min. Na počítači, v době vykonávání testu, běžely pouze základní procesy.

Pro názornost zde jsou uvedeny pouze vybrané výsledky měření. Kompletní výsledky testování se nacházejí spolu se zdrojovými kódy příkladu na přiloženém DVD. Jako první je zhodnocena varianta, kdy se sledovalo pouze vytvoření příslušných struktur bez počítání jakéhokoliv výpočtu.

```
 usecs~error/1        = WALL(us)~e [ rate] USER (us)+ SYS (us) [overhead]
 1,263~9,257/alloc    = 1,26~9,3   [  +0%]  0,5072u +  0,7411s [      0%]
 2,499~1,035/array    =  2,5~1     [ -49%]   1,754u +  0,7201s [     98%]
 1,760~0,854/dsptch_f = 1,76~0,85  [ -28%]  0,9557u +   1,169s [     70%]
 2,062~1,511/dispatch = 2,06~1,5   [ -39%]   1,256u +   1,181s [     95%]
14,688~10,538/fork    = 14,7~11    [ -91%]    2,63u +   13,05s [ 1 156%]
```

Text 1: Výsledky měření výpočtu s jedním prvkem (např. jeden prvek pole).

Výsledky pro případ, kdy je paralelizována pouze jedna operace, dopadají nejlépe pro variantu alokující paměť. Z výsledných časů je vidět, že například vytvoření jednoho bloku a zařazení do jedné fronty je mnohem rychlejší než vytvoření vlákna (řádek s popisem fork). To tráví velmi mnoho času v privilegovaném režimu procesoru a je tak cca. 10 krát pomalejší než varianta `alloc`. Pro varianty s více výpočty už se situace mění.

```
 usecs~error/256      = WALL~err    [rate]   USER +    SYS (us) [overhead]
 0,247~14,911/alloc   = 63,1~3,8e+3 [ +0%] 28,52u +     4,225s [       0%]
 0,790~0,259/array    =  202~66     [-69%] 200,6u +    0,8675s [     515%]
 0,215~0,037/dsptch_f =   55~9,3    [+15%] 82,63u +     3,364s [     163%]
 0,361~0,367/dispatch = 92,5~94     [-32%] 119,3u +      3,53s [     275%]
351,67~15,962/fork    = 9e+4~4,1e+3 [100%] 3 260u + 1,029e+05s [323 989%]
```

Text 2: Výsledky měření výpočtu s 256 prvky.

Ve variantě měření s 256 vytvářenými prvky se vytvoření fronty a zařazení 256 položek reprezentovaných funkcemi stává nejrychlejší variantou. Je rychlejší i než ekvivalent využívající bloky, což je velmi pravděpodobně způsobeno tím, že varianta s bloky kopíruje na

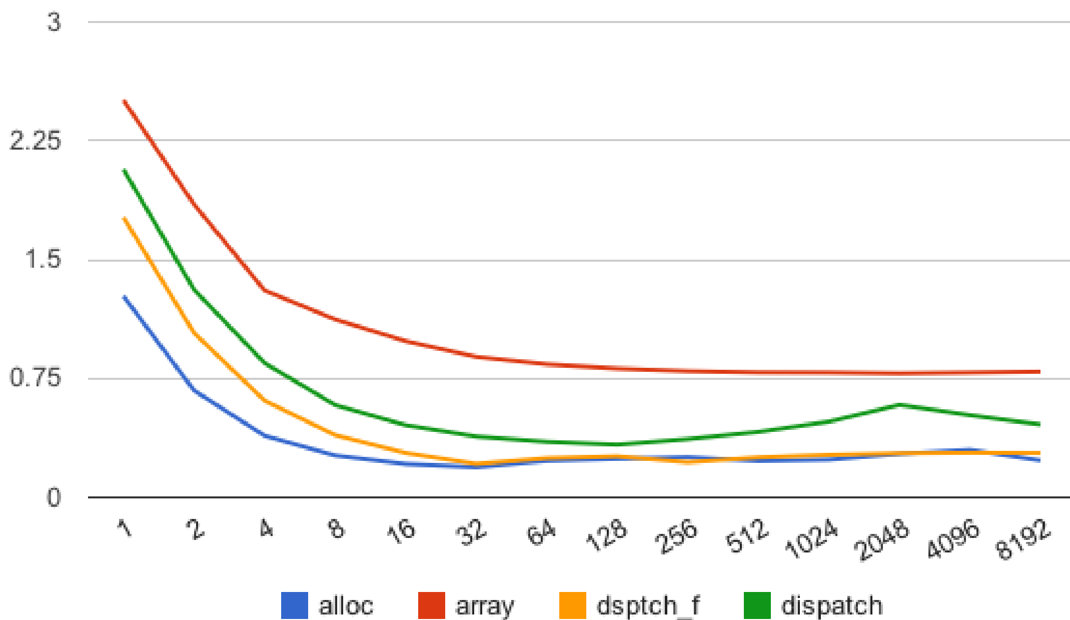
zásobník také proměnné z okolního kontextu a to se ukazuje být velmi pomalou operací. Vytvoření 256 vláken se ukazuje jako extrémně pomalá operace, která je o několik řádů pomalejší než všechny ostatní.

Posledním zmíněným měřením je alokace prostředků pro 4096 prvků. Opět se ukazuje velká efektivita varianty využívající sériovou frontu plněnou objekty reprezentovanými funkcemi, která je nejrychlejší ze všech. Varianta alokace pole je druhá nejrychlejší a za ní je na třetí pozici varianta se sériovou frontou plněnou bloky.

```
usecs~error/4 096     =   WALL~err     [ rate]   USER +      SYS [overhead]
0,293~4,095/alloc     = 1,2e+3~1,7e+4 [  +0%] 497,6u +  57,03s [       0%]
0,781~0,023/array     = 3,2e+3~95     [ -63%] 3 192u +  3,603s [    476%]
0,276~0,250/dsptch_f  = 1,1e+3~1e+3   [  +6%] 1 665u +  4,934s [    201%]
0,512~0,323/dispatch  = 2,1e+3~1,3e+3 [ -43%] 2 546u +  12,69s [    361%]
  523~71,55/fork      = 2,1e+6~2,9e+5 [-100%]   1e5u + 2,3e+6s [436 429%]
```

Text 3: Výsledky měření výpočtu s 4096 prvky.

Výše popsaná měření ukazují na velmi velkou efektivitu práce funkcí knihovny Grand Central Dispatch, která dosahovala nejlepších výsledků spolu s jednoduchou alokací pole. Pořadí jednotlivých variant nebylo vždy zachováno, takže v některých případech byla alokace pole rychlejší než sériová fronta, v jiných případech to bylo naopak. To bylo pravděpodobně způsobeno chybami v měření. Přibližné pořadí výkonnosti však zůstalo stejné.



Obrázek 3.5: Graf rychlosti vytvoření struktur pro paralelní zpracování.

Graf, ze kterého byla pro přehlednost vynechána metoda `fork`, ukazuje, že se stoupajícím počtem prvků klesá doba potřebná pro jeho alokaci. Klesání se projevuje až do určitého limitu (cca 32 prvků), po kterém už doba zůstává konstantní.

Druhá metoda měření se zaměřovala na čas potřebný pro výpočet zmiňované trigonometrické funkce. Pro malý počet položek způsobuje režie související se spuštěním paralelního

zpracování zpomalení, které opět znevýhodňuje hlavně využití vláken. Velmi dobrých výsledků dosahuje také knihovna OpenMP, která dělí počet úloh na několik intervalů a ty přiřadí jednotlivým procesorům.

```
  usecs~error/8        = WALL~error [rate]  USER + SYS     [overhead]
  3,222~0,796/loop     = 25,8~6,4   [ +0%] 24,97u + 0,7889s [      0%]
  2,953~1,057/apply    = 23,6~8,5   [ +9%] 29,98u + 8,612s  [     50%]
  7,868~2,457/serial   = 62,9~20    [-59%]  38,1u + 12,74s  [     97%]
  6,387~4,592/parallel = 51,1~37    [-50%] 46,33u + 33,72s  [    211%]
  5,761~4,083/queues   = 46,1~33    [-44%] 55,57u + 20,67s  [    196%]
  4,524~1,420/openmp   = 36,2~11    [-29%] 36,28u + 17,69s  [    110%]
286,380~79,010/thread  = 2290~630   [-99%] 206,4u + 2 661s  [ 11 034%]
```

Text 4: Výsledky měření výpočtu s 8 prvky.

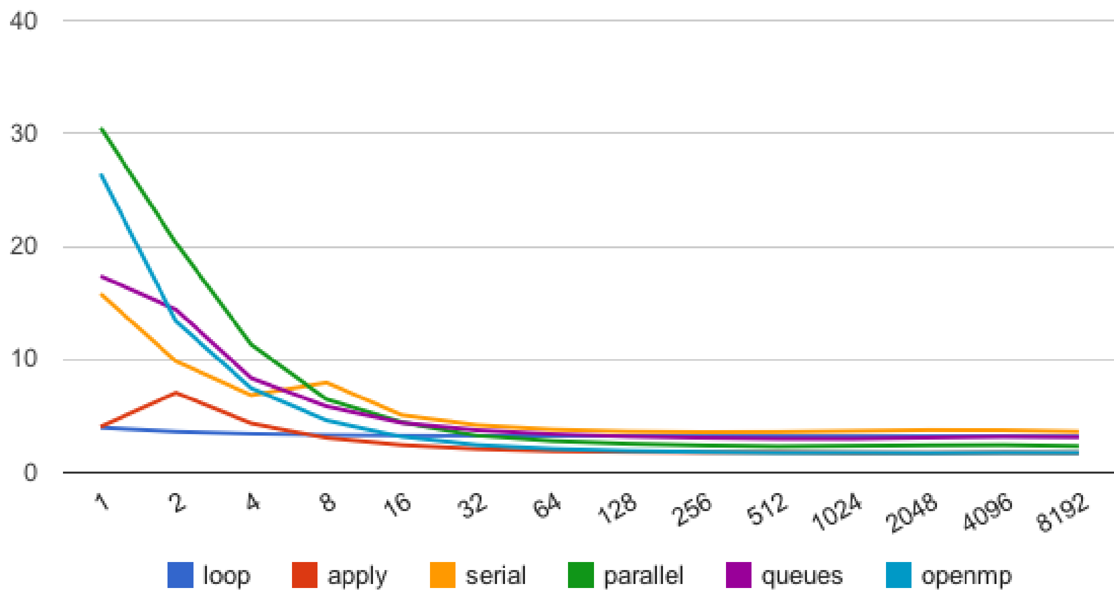
Pro velký počet úloh už se projeví pozitivní důsledky paralelního zpracování úloh. V porovnání se sekvenčním zpracováním dosahují všechny metody lepších výsledku, jen s výjimkou sériové fronty, která je ekvivalentní sekvenčnímu zpracování avšak obsahuje režii navíc. Také využití vláken má stejně jako v ostatních případech velmi špatné výsledky způsobené jejich neefektivním využitím.

```
usecs~error/4 096   =    WALL~error   [rate]  USER (us) +      SYS [overhead]
3,119~0,021/loop    = 1,28e+4~85      [ +0%] 1,276e+04u +   5,922s [      0%]
1,655~0,358/apply   = 6,78e+3~1,5e+3 [+89%] 1,301e+04u +   16,35s [      2%]
3,661~0,764/serial  =  1,5e+4~3,1e+3 [-15%]  1,69e+04u +   46,61s [     33%]
2,321~1,037/parallel= 9,51e+3~4,2e+3 [+34%] 1,822e+04u +   115,3s [     44%]
3,067~1,085/queues  = 1,26e+4~4,4e+3 [ +2%] 2,344e+04u +   217,7s [     85%]
1,654~0,248/openmp  = 6,77e+3~1e+3   [+89%] 1,278e+04u +   33,59s [      0%]
517,4~12,604/thread = 2,12e+6~5,2e+4 [-99%] 7,489e+04u + 2,3e+06s [19 037%]
```

Text 5: Výsledky měření výpočtu s 4096 prvky.

Jak je vidět z výsledků, je nejefektivnějším způsobem paralelního zpracování `for` cyklu funkce `dispatch_apply` určená přímo pro tyto účely a knihovna OpenMP obsahující pro tento případ speciální direktivu. Dobrých výsledků dosáhla i paralelní fronta.

Na následujícím grafu je znázorněna doba obsluhy jedné události. Pro malé počty požadavků se doba zpracování, stejně jako v předchozím případě, velmi liší. Od jisté hodnoty už se doba zpracování neprodlužuje. Křivka sekvenčního zpracování je téměř neměnná a na začátku grafu se jeví jako nejlepší řešení, ale pro větší počty úloh se stává jednou z pomalejších metod.



Obrázek 3.6: Graf rychlosti vytvoření struktur pro paralelní zpracování.

Paralelní programování se ukazuje jako velmi dobrý způsob jak zrychlit provádění výpočtu. Toto měření je pouze ukázkové. Pro reálné výsledky chybí lepší práce s jednotlivými implementacemi. To mohlo způsobit velmi špatné výsledky varianty vytvářející vlákna, která je velmi neefektivní v případě, že pro každou úlohu vytváří nové vlákno. Optimalizací by došlo k jejímu zrychlení. Tato optimalizace by však vedla ke značnému zkomplikování implementace testu. Knihovna OpenMP, která také dosahovala velmi dobrých výsledků, není využitelná pro tolik způsobu paralelizace jako GCD nebo vlákna.

3.11 Zhodnocení knihovny

Jak je vidět v předchozích kapitolách, je knihovna Grand Central Dispatch nejen velmi dobře použitelná, ale také dosahuje velmi dobrých výsledků co se týče výkonnosti. Je využitelná ve většině případů, kde se paralelní zpracování hodí. Ať už se jedná o zpracování úloh na pozadí, práce s nízkoúrovňovými objekty, provedení blokujících operací nebo generování událostí mimo hlavní vlákno aplikace. Ve všech těchto situacích je pro programátora velmi jednoduché s knihovnou pracovat, nemusí řešit složité synchronizační algoritmy a místo toho může použít buď sériovou frontu nebo zámek, který je také knihovna poskytuje. Aplikace se automaticky stává daleko lépe škálovatelnou a udržovatelnou jelikož není nutné řešit kolik vláken je možné vytvořit či kolik je právě dostupných procesorů. Tyto operace jsou kompletně v kompetenci operačního systému. Díky tomu, že datové struktury vláken jsou načteny přímo v operačním systému, nezabírají místo na zásobníku aplikace.

Nevýhodou knihovny je prozatím její nízká přenositelnost. I přesto, že existují porty na jiné systémy, je kvalitní podpora v jádře OS pouze v systému Mac OS X a FreeBSD.

Kapitola 4

HTTP servery

HTTP servery jsou TCP servery pracující s HTTP protokolem. Typicky zpracovávají velké množství požadavků najednou. Webové servery zpracovávají skripty a zobrazují HTML stránky, herní servery rozesílají akce hráčů, FTP servery obsluhují downloady uživatelů atd. Aby bylo možné obsloužit více požadavků najednou a uživatelé nečekali ve frontě, než se zpracují všechny požadavky, které byly do fronty zařazeny před nimi, je nutné tyto požadavky zpracovávat paralelně. K tomu se používají 3 hlavní metody: funkce select, vlákna a procesy.

4.1 Apache

Nejznámější webový server Apache využívá ve verzi 2 tzv. MPM (MultiProcessing Modules). To znamená, že může být nakonfigurován jako čistě procesový server, čistě vláknový nebo jako kombinace předchozích možností.[18] Oficiální verze vydávaná organizací Apache Software Foundation vyžívá hybridní varianty, kde jeden proces spouští řadu dalších procesů, které obsahují velké množství vláken pro obsluhu požadavků. Tato varianta zlepšuje použitím vláken rychlost odezvy, ale zachovává stabilitu díky oddělení vláken do samostatných procesů.[17]

4.2 HTTP protokol

Zkratka HTTP pochází z anglického HyperText Transfer Protokol. Z názvy plyne, že se jedná o síťový protokol běžící na aplikační vrstvě, který slouží pro přenos hypertextových dokumentů. To jsou strukturované dokumenty obsahující hypertextové odkazy. Pomocí těchto odkazů jsou mezi sebou dokumenty vzájemně propojeny. Vývoj protokolu je organizovaný organizací IETF[1]. Tento protokol je v moderních aplikacích používán pro přenos jakýchkoliv typů souborů. To umožňuje rozšíření MIME[2], které bylo původně určeno pro přenos elektronické pošty a později se rozšířilo i do protokolu HTTP. To to rozšíření umožňuje např. přidávat jiné znakové sady než jen ASCII nebo přenášet soubory.

HTTP je založeno na komunikaci požadavek-odpověď a architektuře typu klient-server. Je to tedy bezstavový protokol. Stavovost je emulována na straně aplikace. Klient tedy posílá požadavek serveru, který ho zpracuje a odešle zpět odpověď. Odpověď obsahuje stav

[1] Internet Engineering Task Force
[2] Multipurpose Internet Mail Extensions

dokončení zpracování požadavku spolu s dalšími parametry a tělo zprávy, které obsahuje buď požadovaný zdroj nebo chybovou stránku.

Jak již bylo zmíněno, je HTTP protokolem aplikační vrstvy. Očekává spolehlivou transportní vrstvu pro komunikaci mezi účastníky. Pro tyto účely se používá v drtivé většině příkladu protokol TCP[3]. Pro některá spojení však může být použitý i protokol UDP[4].

4.3 URL

K identifikaci a zjištění umístění dokumentů a služeb slouží tzv. URI[5], což je řetězec s přesně definovanou strukturou. Jeho více využívanou podmnožinou je URL[6]. Jeho druhou podmnožinou je URN[7], které identifikuje zdroj a je neměnné. URL se může měnit spolu se změnou umístění zdroje. Tvar URL je popsaný následujícím způsobem:

```
schéma:hierarchie?parametry#fragment stránky
```

Text 6: Struktura URL.

Schéma určuje zbylý tvar URL. Podle něj se určí jak nakládat se zbytkem URL. Možné hodnoty jsou např. http, ftp, mailto (spuštění e-mailového klienta).

Hierarchie Podle standardu URI může být v libovolném formátu. Jednou z předdefinovaných syntaxí je formát pro umístění zdrojů. V něm po dvojtečce následují dvě lomítka () a název domény nebo IP adresa počítače. Před tím může být ještě uživatelské jméno a heslo ve formátu `jméno:heslo` oddělené zavináčem. Po doméně nebo IP adrese následuje cesta k souboru ve tvaru klasického unixového zápisu.

Parametry Tato část je nepovinná a slouží k předávání dalších parametrů blíže určujících požadovaný zdroj. Nemá standardizovaný formát, ale nejčastěji se používají dvojice `klíč=hodnota` oddělené znakem `&`.

Fragment stránky Určuje bližší umístění v dokumentu.

4.4 HTTP Metody

HTTP 1.1 rozšiřuje sadu základních metod definovaných HTTP 1.0, které je možné se zdrojem provádět. [12]

GET Získání jakékoliv informace. Výchozí metoda při většině operací. Tato akce by neměla mít jiný účinek než vrácení zdroje.

HEAD Identická metoda jako GET, avšak odpověď neobsahuje data, ale pouze hlavičku. Tato metoda se hodí při získávání meta-informací o dokumentu, kdy není potřeba přenášet obsah.

---

[3] Transmission Control Protocol
[4] User Datagram Protocol
[5] Uniform Resource Identifiers
[6] Uniform Resource Locator
[7] Uniform Resource Name

POST metoda je používána pro zaslání nějaké entity na server, který ji zpracuje a odešle zpět odpověď. Typicky je tato metoda použita pro zasílání textu z formulářů, nahrávání souborů nebo vkládání dat přes REST rozhraní.

PUT se používá k úpravě již existujícího objektu, který by na serveru měl existovat. Pokud neexistuje, muže server nový objekt vytvořit, avšak musí o tom klienta informovat zasláním odpovědi typu `201 Created`. Pokud už objekt existoval, je odpověď typu `200 Ok` nebo `204 No Content`.

DELETE maže objekt na serveru, avšak může být přerušena zásahem uživatele. Klient nemusí být obeznámen s výsledkem operace pokud nebyla operace v čase odeslání požadavku dokončena. Úspěšná odpověď na požadavek je `200 Ok` pokud odpověď obsahuje výsledek operace, `202 Accepted` pokud nebyla akce vymazání zatím provedena, nebo `204 No Content` pokud byla akce provedena, ale odpověď neobsahuje žádná data.

4.5 Rozdíl mezi HTTP 1.0 a 1.1

4.5.1 Rozšiřitelnost

HTTP 1.1 poskytuje mnohem větší možnosti rozšiřitelnosti. Jelikož před vydáním specifikace HTTP 1.1 implementovalo mnoho výrobců svá proprietární řešení, docházelo k velkým problémům s kompatibilitou. HTTP 1.1 tak muselo být s těmito verzemi kompatibilní i navzdory tomu, že některé z nich obsahovaly chyby. To vedlo k mnoha nelogickým a nesouvislým částem nové specifikace. Je to však součástí vývoje specifikace. Protokol HTTP 1.1 tak může obsahovat jakékoliv hlavičky požadavků a v případě, že server hlavičce požadavku nerozumí, musí ji ignorovat. Díky tomu je zajištěna maximální možná rozšiřitelnost beze změny protokolu.

4.5.2 Cache

Další odlišností HTTP 1.1 je využití cache. HTTP 1.0 poskytuje jednoduchý způsob uchování v cache pomocí parametru `Expires` v hlavičce. Ten určuje jak dlouho může cache posílat stejný dokument. Po uplynutí této doby cache zažádá původní server tzv. podmíněným požadavkem s parametrem `If-Modified-Since` v hlavičce. Pokud se dokument stále nezměnil, odpovídá server stavovým kódem `304 Not Modified`, jinak posílá odpověd se stavem `200 Ok` s novým dokumentem, který nahradí ten původní. HTTP 1.0 také poskytuje parametr hlavičky `Pragma: no-cache`, který indikuje, že požadovaný soubor by neměl být posílaný z cache, ale z původního umístění. Tento koncept cache pracoval dobře, ale měl několik omezení. Nedával serverům nebo klientům plnou kontrolu nad cache. To vedlo k problémům, kdy některé dokumenty byly ukládány do cache i přesto, že neměly být a jiné byly zase nebyly v cache i když do ní patřily.

V terminologii HTTP 1.1 je záznam v cache „čerstvý“ dokud nepřekročí čas pro jeho vypršení. Pak se stává „prošlý“. Cache ho nemusí zrušit, ale před odesláním takového záznamu klientovi ho musí znovu ověřit na původním serveru. Protokol však umožňuje toto chování předefinovat jak pro server, tak pro klienta. Oproti HTTP 1.0, které používá časový identifikátor s přesnosti na jednu sekundu používá HTTP 1.1 takzvaný `entity tag`. Pokud mají dvě odpovědi na požadavky stejný `entity tag`, pak musí být požadované objekty shodné. Server může `entity tag` generovat podle informací, které si zvolí. Může to být například

ukazatel v databází, časová známka nebo jiné hodnoty, které však musí být vždy unikátní. Pro efektivnější dotazování poskytuje HTTP 1.1 nové podmíněné požadavky. Základem je `If-None-Match`, které umožňuje klientovi ověřit jeden nebo několik `entity tagů`. Pokud ani jeden z ověřovaných tagů nesouhlasí se současným tagem souboru, vrátí server normální odezvu, jinak může vrátit hlavičku se stavem `304 Not Modified` s `ETag` parametrem, indikujícím který záznam v cache je platný.

4.5.3 Přenos dat

Protože limit objemu přenesených dat po síti je vždy omezen, je vhodné omezit přenos na minimum. HTTP 1.0 neposkytovalo žádné prostředky pro minimalizaci objemu datového toku. Pokud např. klient potřeboval v odpovědi pouze HTTP hlavičku, nebyl žádný způsob jak toho dosáhnout a vždy musel přijmout celý soubor. Jiným příkladem plýtvání byl případ, kdy se na server odesílal soubor, který přesahoval maximální možnou velikost, kterou mohl server přijmout. Server neměl jak klientovi oznámit, že soubor odmítne a musel ho celý načíst. Chyběla podpora pro vyjednávání klienta se serverem, zda je server požadavek přijme.

Pokud klient požaduje pouze část dokumentu nebo chce pokračovat ve stahování souboru, které bylo přerušeno v půlce, dovolují HTTP 1.1 požadavky specifikovat rozsah části zdroje, kterou požaduje. Rozsah se specifikuje pouze v bytech a uvádí se v parametru `Range`. Server může odpovědět buď celým souborem a ignorovat požadavek na rozsah, nebo může odeslat zpět pouze požadované rozsahy, kterých může být požadováno i více než pouze jeden.

Odpověď, která obsahuje pouze část obsahu je zasílána se stavem `206 Partial Content`, který zabraňuje cache serverům podporujícím pouze HTTP 1.0 v ukládání částí jako celého souboru. Začátek a konec jednotlivých částí reprezentuje parametr `Content-Range` v hlavičce, který spolu s novým MIME typem `multipart/byteranges` umožňuje přenést více částí obsahu v jedné zprávě.

Jak již bylo zmíněno výše, není vždy žádoucí zasílat celý obsah souboru, pokud může nastat situace, že server zprávu nepřijme. Pro tyto potřeby byl v HTTP 1.1 zaveden nový stavový kód `100 Continue` pro oznámení klientovi, že může přenést tělo zprávy. Při použití tohoto mechanizmu klient nejprve pošle hlavičku zprávy a čeká na potvrzení. Po jeho příjmu odešle zbytek požadavku. Pokud klient dostane zpět např. zprávu se stavem `401 Unathorized`, neposílá už dále nic.

4.5.4 Komprese přenášených dat

I když je většina obsahu na webu již komprimována (JPG, GIF atd.), existuje stále obsah, který komprimovaný není a jehož komprimaci se podle studií dá dosáhnout úspory až 40%. HTTP 1.1 rozlišuje kódování end-to-end[8] přenosů a hop-by-hop.[9] Kvůli tomuto rozlišení přidává parametr `Transfer-Encoding`, které určuje kódování v průběhu `hop-by-hop` přenosu. Kódování přenosu `end-to-end` obsahuje už HTTP 1.0 v parametru `Content-Encoding`, který verze 1.1 také používá.

[8] Přenos ze serveru ke klientovi.

[9] Přenos částí výsledného souboru mezi jednotlivými síťovými uzly na cestě klientovi.

4.5.5 Jméno serveru

Vzhledem k tomu, že při návrhu HTTP 1.0 se nepočítalo s tím, že by na jednom serveru mohlo běžet více domén, nastal po masivním rozvoji internetu a prodeji doménových jmen problém. Požadavky v HTTP 1.0 totiž obsahují pouze cestu k dokumentu a server určuje IP adresa a port počítače na kterou je požadavek zaslán. Proto nemohlo na jednom server běžet více domén najednou. DNS však umožňuje přiřadit jednu IP adresu více doménovým jménům. Protože kvůli zpětné kompatibilitě nebylo možné změnit formát řádku s metodou a požadovaným souborem, zavedla skupina IESG[10] v protokolu HTTP 1.1 nový parametr `Host`, který obsahuje hostname a popř. také port ke kterému požadavek patří. Jelikož klienti pracující s HTTP 1.0 nezasílají tento parametr, musí HTTP 1.1 servery odmítat všechny HTTP 1.1 požadavky, které tento parametr neobsahují. Tím je zajištěno, že požadavky od HTTP 1.0 klientů nebudou odmítnuty, ale server se je pokusí zpracovat.

4.5.6 Další rozdíly

Dalšími rozdíly, kterými se nová verze HTTP protokolu liší od předchozí jsou nové chybové kódy. Původních 16 rozšiřuje o dalších 24 dalších možných stavů a přidává i podporu pro varování obsažená v hlavičce odpovědi. Ty mohou obsahovat informace o neočekávaném chováni a dalších chybách, které sice nebrání v provedení požadavku, ale mohou ho ovlivnit.

Nová verze HTTP protokolu dále obsahuje vylepšení autentizace, a bezpečnosti. Poskytuje například parametr `Referer`, který brání neočekávaným přístupům ke stránce. Dále je např. možné zasílat MD5 hash stránky, který umožňuje ověřit identitu stránky, zda nebyla při transportu sítí pozměněna.

Pro servery nabízející vícejazyčný obsah slouží v hlavičce parametry `Accept-Language` a `Accept-Charset`. To jsou parametry zasílané klientem, podle kterých může server zvolit příslušnou jazykovou variantu. Parametr specifikující jazyk může obsahovat také priority podle kterých se volí, které jazyky zasílat. Pokud server dostane požadavek na zdroj, který se může měnit, posílá zpět stavový kód `300 Multiple Choices`, který obsahuje seznam variant s jejich popisem. Klient se rozhodne kterou variantu zvolí a ta je mu následně zaslána.

Jak je vidět, liší se obě specifikace ve velmi mnoha vlastnostech. Většina z nových vlastností je pro dobro věci, avšak některé byly uvedeny aniž by byly vyzkoušeny v reálné praxi. Specifikace HTTP 1.1 se oproti verzi 1.0 na délku ztrojnásobila. Obsahuje také velmi mnoho nepravidelností kvůli kompatibilitě s předchozí verzí. Díky snadnému rozšíření však umožní snadné uvedení dalších modifikací nebo verzí. [6]

[10]Internet Engineering Steering Group je skupina řídící organizaci IETF.

Kapitola 5

Antivirový software AVG

Produkty společnosti AVG chrání počítač na několika úrovních. AVG Anti-Virus a Anti-Spyware zajišťují ochranu proti virům a spyware. Další produkty chrání počítač před hackery, nevyžádanými e-maily, nebezpečnými stránkami a soubory na internetu a také před nebezpečími na sociálních sítích.

5.1 Metody detekce

Účinnost produktů AVG v oblasti detekce infikovaných souborů a programů typu exploit je dána vícevrstvou ochranou. Aby bylo dosaženo rychlejšího prověřování, jsou soubory předzpracovány a jsou vyloučeny oblasti, ve kterých není nutné provádět virovou analýzu.[5]

Detekce založená na signaturách Tato technika srovnává soubory se signaturami známých virů. Signatura je řetězec bajtů, který je pro daný vir specifický. Poté je provedená podrobná analýza, aby byl identifikován přesný typ infekce.

Detekce založená na polymorfech Tato metoda se běžně používá pro detekci známých virů a pro určení nových variant již rozpoznaných virů. Polymorfní detekce vyhledává charakteristické sekvence určitých virů. Takové sekvence se obvykle nemění, i když je vir upraven, a to dokonce ani v případě, kdy nová varianta vykazuje odlišné chování. Tato metoda je účinná hlavně při detekci makrovirů a skriptových virů.

Analýza na základě heuristiky Třetí vrstvou detekce virů je heuristická analýza, která sleduje chování softwaru a zjišťuje, zda je škodlivý. To umožňuje detekovat i viry, které nejsou zahrnuty v interní databázi virů. Využívány jsou dvě hlavní metody:

- Statická heuristická analýza hledá podezřelé datové konstrukce.
- Dynamická heuristická analýza emuluje činnost kódu v chráněném prostředí virtuálního počítače softwaru AVG.

Analýza na základě chování Čtvrtou vrstvou detekce virů je analýza chování. Tato technologie (s projednávaným patentem) sleduje, co software při spuštění dělá. Tato technologie pomocí různých klasifikátorů a pokročilých algoritmů odhaluje škodlivé chování souborů a zabraňuje jejich spuštění.

5.2 Architektura

Architektura systém AVG Anti-Virus pro operační systémy Linux a FreeBSD se skládá z několika samostatně běžících procesů. Tyto procesy obsahují další vlákna, která mezi sebou navzájem komunikují. Mezi hlavní procesy patří:

WatchDog je hlavní proces, který monitoruje, hlídá a spravuje všechny ostatní procesy.

Scheduler se stará o naplánované úlohy a aktualizace programu, antispamové a antivirové databáze.

Tcpd E-mailový filtr pro skenování phishingu, virů nebo spamu.

Avgdump je automaticky spouštěn v případě, že proces AVG neočekávaně skončí. Jeho účelem je vygenerovat informace o pádu procesu, prověřit konfiguraci a v případě, že je to možné, tak uložit záznam do systémového logu.

Oad Tento proces kontroluje soubory při přístupu k nim. V případě, že soubor obsahuje vadný kód, je přístup k němu odepřen.

Scan je rozhraní v příkazové řádce k procesu `avgscand` sloužící jako antivirový skener.

5.3 Využití GCD

V dnešní době už antivirový software neslouží pouze pro kontrolu souborů přinesených na disketách nebo CD, ale komplexně chrání celý systém. Prohledává soubory, se kterými systém pracuje, kontroluje e-mailové přílohy, nebo hlídá podvržené odkazy a navštívené stránky v prohlížeči. Je proto velmi náročný na systémové prostředky a tak musí být veškeré operace, kterých může v jeden okamžik probíhat velké množství, být implementovány maximálně efektivně, aby antivirový software zbytečně nebrzdil celý systém.

Knihovna Grand Central Dispatch se v tomto případě jeví jako ideální způsob, jak implementovat jednotlivé operace jakými může být prověření jednoho souboru na disku nebo přílohy e-mailu, zkontrolování odkazu na webové stránce nebo stažení aktualizace virové databáze. Jelikož se jedná o na sobě nijak nezávislé operace, je každou možné provádět například v globální paralelní frontě. Ušetří se tak systémové prostředky a zvýší rychlost.

Nevýhodou je přenositelnost na ostatní platformy. Jelikož je AVG multiplatformní, musela by se vyřešit kompatibilita s ostatními systémy. K tomu by mohly posloužit implementace GCD API, které se začínají objevovat i pro Linux a Windows. Jejich spolehlivost a výkonnost je však prozatím neotestovaná.

Kapitola 6

Implementace HTTP serveru

Pro demonstraci možností a výhod knihovny Grand Central Dispatch byl zvolen jednoduchý HTTP server napojený na antivirový software AVG. Využívá většiny možností knihovny. Jeho hlavní částí je volitelná metoda paralelního zpracování příchozích požadavků nastavená při spuštění. Server je poměrně snadno rozšiřitelný o další metody zpracování a jednoduše konfigurovatelný v konfiguračním souboru. Ten je zpracován pomocí třídy `ConfigFile` jejímž autorem je Benjamin Reh[1]. Server umožňuje prověření požadovaných souborů antivirovým programem AVG. K tomu využívá proces AVG Tcpd, který slouží e-mailovým serverům k ověřování e-mailových příloh.

6.1 Metody zpracování požadavků

Jelikož HTTP servery musí zpracovávat velké množství požadavků najednou, je nutné je zpracovávat paralelně, jinak by došlo k zablokování programu. Pro tyto účely se běžně používají vlákna, která jsou podporována napříč všemi operačními systémy a servery jsou tak snadno přenositelné. Pro demonstrační účely byla u implementovaného serveru dále zvolena metoda vytváření nových procesů funkcí `fork` a knihovna Grand Central Dispatch. Tyto metody je možné při startu serveru měnit a testovat tak jednotlivé varianty.

Při příchodu nového požadavku je funkci, která zpracovává tuto událost předán ukazatel na funkci, která reprezentuje metodu zpracování. Veškeré informace o příchozím spojení jsou dostupné ve struktuře typu `reqInfo`. Ta obsahuje socket na kterém jsou data dostupná (proměnná `socket`) a údaje potřebné pro navázání komunikace s AVG. Tato struktura je předána do společné funkce, která obsahuje celou část vykonávající se paralelně. To znamená načtení příchozího požadavku, jeho zpracování a odeslání odpovědi nazpět klientovi.

Implementace jednotlivých způsobů paralelizace jsou co nejjednodušší a neřeší žádné dodatečné úkoly. Je tak názorně vidět rozdíl jednotlivých v náročnosti jednotlivých variant.

Při paralelizaci se pouze zavolá funkce `processHttpRequest` a po jejím provedení se vlákno nebo proces ukončí. Aby v případě varianty vytvářející nové procesy nedocházelo k zahlcení systému zombie procesy, je signál `SIGCHLD` ignorován. Paralelní zpracování s pomocí vláken je v tomto případě řešeno poměrně neefektivně a to tak, že pro každý požadavek je vytvořeno nové vlákno. V případě reálného nasazení by bylo vhodné použít složitější způsob práce s vlákny, kterým je například tzv. „thread pool“, kde proces spravuje několik vláken, kterým předává úlohy podle toho, jak jsou jednotlivá vlákna vytížena. Implementace tohoto způsobu by však značně zkomplikovala a znepřehlednila celý kód serveru.

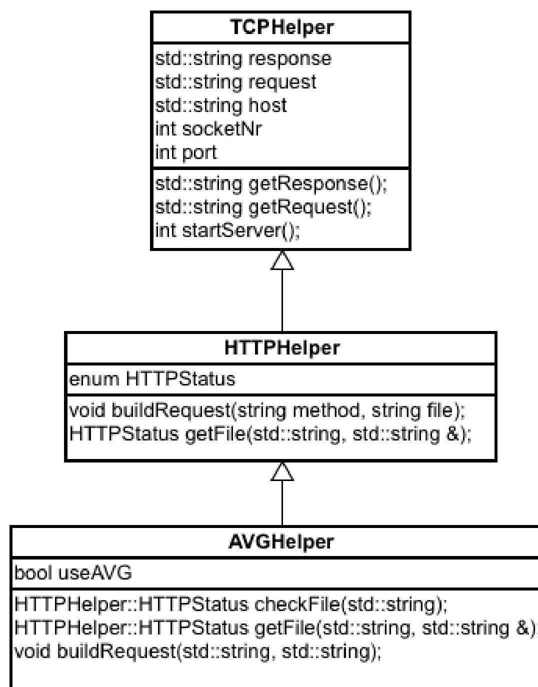
[1] https://github.com/benreh/configfile

Další metodou paralelizace je vytvoření samostatné sériové fronty pro každý požadavek. Jelikož sériové fronty mohou běžet paralelně vedle sebe je i tento přístup možný. Vzhledem k tomu, že zde dochází k velkému plýtvání prostředky na neustálé vytváření nových front, není tato metoda vhodná a je uvedena pouze pro ilustraci možného přístupu.

Poslední metodou je využití globální paralelní fronty poskytované knihovnou Grand Central Dispatch do které je vložen blok s voláním funkce `processHttpRequest`. Jelikož se jedná o globální frontu, není nutné ji explicitně rušit nebo uvolňovat. Její obsluha je kompletně v roli operačního systému.

6.2 Architektura aplikace

Pro implementaci byl zvolen jazyk C++. I když se v systému Mac OS X využívá spíše Objective-C, je díky C++ v budoucnu možné spustit server i v operačním systému FreeBSD. V aplikaci se nachází 3 na sobě závislé třídy. `TCPHelper` implementuje základní funkce pro přenos dat pomocí TCP. Nachází se v ní také implementace serveru, proto pro inicializaci serveru stačí zavolat pouze metoda `startServer`. Od této třídy dědí třída `HTTPHelper`, která rozšiřuje původní třídu o metody práce s HTTP požadavky. Poslední třídou je `AVGHelper`, který přetěžuje některé metody třídy `HTTPHelper` a přidává podporu pro ověření požadovaných souborů antivirovým softwarem AVG.



Obrázek 6.1: Zjednodušený diagram tříd.

Na obrázku 6.1 se nachází zjednodušený diagram tříd ukazující atributy a některé vybrané metody.

6.3 Počítadlo požadavků

Server má pro demonstrační účely implementováno globální počítadlo požadavků které přijal a na které odpověděl. V případě, že se klient v průběhu zpracovávání požadavku odpojí, nezasílá mu server žádnou odpověď a pokračuje v dalším příjmu nových požadavků. Není tedy ani navýšeno počítadlo odpovězených požadavků. Vzhledem k tomu, že zpracovávání probíhá paralelně a do počítadla je neustále zapisováno, mohlo by docházet ke kolizím a přepisům od jednotlivých vláken.

Sériová fronta v tomto případě slouží jako velmi jednoduchý a účinný způsob jak synchronizovat přistup do kritické sekce, kterou je zápis do sdílené proměnné. Vzhledem k tomu, že úlohy jsou v sériové frontě zpracovávány postupně, je zaručeno, že nemůže nikdy dojít k současnému zápisu ze dvou různých bloků a tím k přepsání hodnoty proměnné neplatnou hodnotou. Odpadá tak tak implementace zámků a je nahrazena jednoduchým a rychlým řešením.

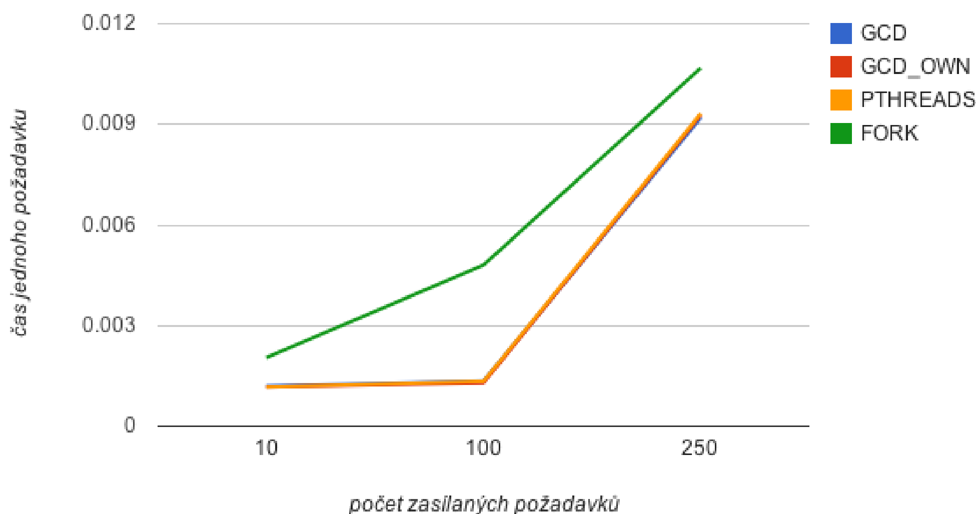
Počítadlo požadavků je pravidelně vypisováno na standardní výstup. K tomuto účelu je využita další součást knihovny Grand Central Dispatch, kterou je `dispatch source` nastavený na periodické generování událostí. K výpisu se používá globální sériová fronta odlišná od fronty inkrementující počet přijatých a zodpovězených požadavků. Interval výpisu je možné nastavit v konfiguračním souboru. Jako callback časovače je nastavena funkce `dispatchPrintStatus`, která pouze vypíše informace o počtu požadavků z globálních proměnných na standardní výstup.

Pro variantu, která zpracovává požadavky v nových procesech je inkrementování čítačů požadavků deaktivováno z důvodu složité mezi-procesové komunikace, která není hlavním předmětem implementace.

6.4 Testování serveru

Pro testování výkonu a paralelizace serveru slouží skript `test.rb`. Jedná se o program v jazyce Ruby, který vytvořením velkého množství vláken simuluje reálné zatížení serveru. počet požadavků, které jsou serveru poslány je možné specifikovat jako parametr příkazové řádky. Jelikož je počet otevřených souborů (v tomto případě socketů) omezen, je ve skriptu implementován semafor dovolující pouze 250 otevřených spojení. Tím je zajištěno, že skript neskončí chybou v případě jejich vyčerpání je možné testovat takřka neomezené množství současně vyslaných požadavků. Reálně jich však bude vždy maximálně 250. Každé měření je opakováno několikrát a na konci je zobrazen nejlepší a průměrný výsledek.

Výsledky testování jsou zatíženy neznámou chybou, která při větším počtu zároveň zaslaných požadavků způsobovala mnohonásobně delší dobu některých odpovědí. Příčina chyby nebyla zjištěna. Výsledky testů jsou proto uváděny pro nejlepší výsledek a tím je náhodné zdržení eliminováno. Maximální počet požadavků pro test byl zvolen 250 z důvodu implementace zámku, která spotřebuje režii navíc a tím by mohla ovlivnit výsledky testu.



Obrázek 6.2: Výsledky testu výkonnosti serveru.

Z výsledků plyne, že pro takto malé počty požadavků se rozdíly mezi variantami smazávají a jsou zanedbatelné. Výjimkou je pouze varianta `FORK`, která dosahuje velmi malé efektivity.

6.5 Antivirová kontrola souborů

Na serveru je implementovaná antivirová kontrola požadovaných souborů. Kontrola je prováděna před načtením souboru ze složky serveru a jeho odesláním. V případě, že je soubor infikován je klientovi vrácen chybový stav `403 Forbidden` a chybová stránka `403.html` z interní složky serveru.

Pro antivirovou kontrolu je využíván proces AVGTcpd sloužící pro kontrolu e-mailových příloh. Jeho rozhraní je dostupné přes socket. Standardně je spuštěn na portu 54322. Pro kontrolu se tak stačí vytvořit spojení na `localhost:54322` a poslat správný požadavek.

Požadavek se zasílá ve tvaru: `scan název_souboru` a musí být zakončen znakem nový řádek. Název souboru musí obsahovat celou cestu k souboru od kořenového adresáře. V případě, že je soubor v pořádku, vrací se odpověď `200 ok`. Pokud je soubor infikovaný, je vrácena odpověď ve tvaru: `403 File 'TUNE.VBS' infected: 'Virus found JS/Generic'`. Stavový kód 403 označuje, že požadavek klienta byl sice správný, ale z nějakého důvodu byl přístup k požadovanému souboru odmítnut.

Implementace kontroly přes socket bylo možné provést dvěma způsoby. Prvním způsobem je vytvoření spojení a zasílání požadavků pomocí jednoho socketu. Výlučný přístup by se dal zajistit globální sériovou frontou, kterou poskytuje knihovna GCD. V případě uváznutí jednoho požadavku, či kontrole velkého souboru by však byly zablokovány ostatní požadavky ve frontě a mohlo by dojít i k uváznutí. Výhodou tohoto řešení je malá režie na navázání spojení s démonem Tcpd. Druhým způsobem, který byl implementovaný, je nové navázání spojení pro každý kontrolovaný soubor. Vzniká tak velká režie při neustálém navazování a rušení spojení, ale dochází k eliminaci zpoždění při čekání ve frontě. Nemůže dojít ani k uváznutí více požadavků, jelikož všechny kontroly jsou na sobě zcela nezávislé.

6.6 Nastavení serveru

Nastavení serveru se provádí v konfiguračním souboru `config.cfg`, který musí být umístěn ve stejném adresáři jako spustitelný soubor serveru. V tomto souboru je možné nastavit:

document_root obsahující cestu ke složce ve které jsou umístěny veřejné soubory serveru. To znamená všechny soubory dostupné uživateli.

internal_root obsahující cestu ke složce s interními soubory serveru. Typicky jsou to chybové stránky, které se uživateli odesílají v případě nenalezení souboru nebo odhalení virové infekce.

port je číslo portu, na kterém je server dostupný pro komunikaci.

info_interval určující po jakém intervalu vypisovat na standardní výstup informaci o počtu přijatých a zodpovězených požadavků. V případě, že je nastaven na 0, nejsou vypisovány žádné informace.

avg_check povoluje, resp. zakazuje kontrolu odchozích souborů na přítomnost virové infikace. Pokud je nalezen v souboru virus, je poslána chybová stránka spolu s chybovým kódem v HTTP hlavičce.

avg_host je adresa počítače na kterém je dostupný AVG Tcpd démon kontrolující soubory.

avg_port označuje číslo portu, na kterém přijímá AVG Tcpd požadavky od ostatních programů.

Kapitola 7

Závěr

Technologie Grand Central Dispatch se v této práci ukázala jako velmi jednoduchý a efektivní způsob paralelizace programu. Režie, která je s ní spojena je oproti ostatním metodám velmi malá, takže je možné ji použít i u méně náročných úloh, kde se hledí na prostředky spotřebované vytvářením všech potřebných struktur. Tím získává velkou výhodu oproti vláknům, která by měla tato knihovna nejčastěji nahradit. API knihovny je velmi jednoduché a tak se programátor už nemusí soustředit na to jak paralelní zpracování úloh vyřešit, ale pouze na to jak ho co nejvíce využít. Nemusí řešit ani synchronizační problémy, jelikož pro ně GCD poskytuje velmi účinné mechanismy, které jsou využitelné pro většinu situací.

Při implementaci serveru se také projevily výhody GCD oproti ostatním implementacím. Implementace s procesy se mezi sebou obtížně synchronizovaly a komplikace nastaly u počítadla požadavků, které by muselo být implementováno pomocí složité meziprocesové komunikace. Varianta využívající Grand Central Dispatch dosahovala vysoké efektivity a jednoduchosti.

Použití zkoumané knihovny tak nezbývá než doporučit, jelikož z paralelního programování dělá velmi snadnou záležitost, kterou zvládne i programátor, který nezná všechny nástrahy plynoucí z paralelizace. Jedinou nevýhodou knihovny je její omezení na malé množství operačních systémů, které podporuje. Díky uvolnění zdrojových kódů by se však knihovna měla v brzké době rozšířit i na další systémy.

Paralelní programování však není vhodné využít kdekoliv. Programátor se musí zamyslet nad režií navíc, kterou si inicializace potřebných struktur vyžádá. Ta může u jednoduchých úloh být větší než náročnost celého výpočtu. Je proto nutné využívat této technologie s rozvahou.

Literatura

[1] Core Foundation. [online], 2010 [cit. 2010-12-21].
URL http://developer.apple.com/corefoundation/

[2] Grand Central Dispatch (GCD) Reference. [online], 2010 [cit. 2010-12-21].
URL http://developer.apple.com/library/mac/documentation/Performance/Reference/GCD_libdispatch_Ref/Reference/reference.html

[3] Kernel Programming Guide - BSD Overview. [online], 2010 [cit. 2010-12-21].
URL http://developer.apple.com/library/mac/documentation/Darwin/Conceptual/KernelProgramming/BSD/BSD.html

[4] Concurrency Programming Guide. [online], 2010 [cit. 2010-12-9].
URL http://developer.apple.com/library/mac/documentation/General/Conceptual/ConcurrencyProgrammingGuide/Introduction/Introduction.html

[5] Přehled technologií. [online], 2011 [cit. 2011-01-09].
URL http://www.avg.com/cz-cs/avg-software-technology

[6] Balachander Krishnamurthy, D. M. K., Jeffrey C. Mogul: Key Differences between HTTP/1.0 and HTTP/1.1. [online], 1999 [cit. 2011-05-18].
URL http://www2.research.att.com/~bala/papers/h0vh1.html

[7] Barney, B.: Introduction to Parallel Computing. [online], 2010 [cit. 2010-12-28].
URL https://computing.llnl.gov/tutorials/parallel_comp/

[8] Dvořák, V.: Architektury a programování paralelních systémů. [online], [cit. 2011-01-08].
URL https://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/ARC-IT/lectures/ARC08.pdf

[9] Hakes, J.; Evans, B.: Apple Previews Mac OS X Snow Leopard to Developers. [online], 2008 [cit. 2010-12-11].
URL http://www.apple.com/pr/library/2008/06/09snowleopard.html

[10] Jepson, B.; Rothman, E. E.: 5.2. The System Library: libSystem. [online], 2003 [cit. 2011-05-21].
URL http://docstore.mik.ua/orelly/unix3/mac/ch05_02.htm

[11] Kašpárek, T.; Kočí, R.; Peringer, P.; aj.: Operační systémy, Studijní opora. [online], 31.05.2010 [cit. 2011-01-02].
URL http://www.fit.vutbr.cz/study/courses/IOS/public/IOS-opora.pdf

[12] kolektiv autorů: Hypertext Transfer Protocol – HTTP/1.1. [online], 1999 [cit. 2011-05-08].
URL http://www.w3.org/Protocols/rfc2616/rfc2616.html

[13] Lampa, P.: Procesy a vlákna, přepínání kontextu. [online], 15.03.2007 [cit. 2011-01-02].
URL https://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/POS-IT/lectures/2007/os2p07-03.pdf

[14] Marius Zwicker: libXDispatch - Overview. [online], 2011 [cit. 2011-04-28].
URL http://opensource.mlba-team.de/xdispatch/index.html

[15] Romanchenko, V.: Evolution of the multi-core processor architecture Intel Core: Conroe, Kentsfield... [online], 27.06.2006 [cit. 2010-12-28].
URL http://www.digital-daily.com/cpu/new_core_conroe/

[16] Siracusa, J.: Mac OS X 10.6 Snow Leopard: the Ars Technica review. [online], 2009 [cit. 2011-05-21].
URL http://arstechnica.com/apple/reviews/2009/08/mac-os-x-10-6.ars/12

[17] The Apache Software Foundation: worker - Apache HTTP Server. [online], [cit. 2011-01-05].
URL http://httpd.apache.org/docs/2.0/mod/worker.html

[18] The Linux Documentation Project: Apache Overview HOWTO: Apache. [online], [cit. 2011-01-05].
URL http://tldp.org/HOWTO/Apache-Overview-HOWTO-2.html

[19] van Vechten, J.: Introducing blocks and Grand Central Dispatch on iPhone. [online], 2009 [cit. 2011-05-21].
URL http://developer.apple.com/videos/wwdc/2010/

[20] Vlastimil Král: Přehled technologií. [online], 2011 [cit. 2011-04-28].
URL http://www.mujmac.cz/art/sw/gcd-pod-freebsd-21-10-09.html

[21] Watson, R. N. M.: [libdispatch-dev] GCD MPM for Apache. [online], 2010 [cit. 2011-05-03].
URL http://lists.macosforge.org/pipermail/libdispatch-dev/2010-May/000352.html

Příloha A

Obsah DVD

- Zdrojové kódy implementovaného HTTP serveru.
- Zkompilovaná verze HTTP serveru.
- Zdrojové kódy této práce.
- Zdrojové kódy aplikace na porovnání výkonnosti.
- Kompletní výsledky testu výkonnosti.
- Antivirový software AVG.