VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

IMPLEMENTACE GRAND CENTRAL DISPATCH V C++

GRAND CENTRAL DISPATCH IMPLEMENTATION FOR C++

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **MAREK ŠALGOVIČ**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. MARTIN HRUBÝ, Ph.D.**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2019/2020

Zadání bakalářské práce

Student: **Šalgovič Marek**
Program: Informační technologie
Název: **Implementace Grand Central Dispatch v C++**
Grand Central Dispatch Implementation for C++
Kategorie: Operační systémy

Zadání:

1. Prostudujte paralelní programování podle Grand Central Dispatch (GCG) a knihovny C++ (STL, Boost).
2. Navrhněte obdobu GCD pro C++. Zaměřte se na efektivní využití stávající podpory z knihoven STL a Boost. Zaměřte se na vysokou míru abstrakce kódu.
3. Implementujte knihovnu.
4. Knihovnu testujte na sadě ukázkových příkladů v prostředích s různým počtem procesorů. Vyhodnoťte výpočetní efektivitu knihovny.

Literatura:

- Technická dokumentace Grand Central Dispatch firmy Apple.
- Dokumentace standardu C++.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Hrubý Martin, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 28. května 2020
Datum schválení: 5. dubna 2020

Abstrakt

Grand Central Dispatch je systém, ktorý aplikáciám umožňuje optimálne využitie viacjadrových zariadení od firmy Apple. Tento systém je veľkou časťou podporovaný operačným systémom Apple platforiem. Cieľom tejto bakalárskej práce bolo analyzovať systém a následne navrhnúť a implementovať obdobu v podobe knižnice v jazyku C++. Táto knižnica poskytuje rozhranie a funkcionalitu podobnú existujúcemu systému.

Abstract

Grand Central Dispatch is a system which allows applications to optimally use multi-core Apple devices. This system is for the most part supported by operating systems of Apple platforms. The goal of this bachelor's thesis was to analyze the system and subsequently design and implement a library in C++. This library provides interface and functionality similar to the existing system.

Kľúčové slová

vlákno, viacvláknové programovanie, Grand Central Dispatch, C++, Apple, pthreads, POSIX

Keywords

thread, multithreading, Grand Central Dispatch, C++, Apple, pthreads, POSIX

Citácia

ŠALGOVIČ, Marek. *Implementace Grand Central Dispatch v C++*. Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Martin Hrubý, Ph.D.

Implementace Grand Central Dispatch v C++

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne, pod vedením pána Ing. Martina Hrubého, Ph. D. a uviedol som všetky literárne zdroje, publikácie a iné zdroje, z ktorých som čerpal.

.....
Marek Šalgovič
28. mája 2020

Poďakovanie

Rád by som poďakoval pánovi Ing. Martinovi Hrubému, Ph.D. za odborné vedenie práce, za vysvetlenie problematiky pri začiatku práce, usmernenie a doladenie nedostatkov pri návrhu a kontrolu implementácie a práce celkovo.

Obsah

Kapitola 1

Úvod

Postupom času sa výpočetná technológia zlepšuje a jedno zariadenie má viacero procesorov alebo jadier, ktoré sú schopné vykonávať inštrukcie paralelne. Programy môžu využívať tieto procesory na vykonávanie úloh paralelne resp. súbežne. Využitie týchto procesorov je pri dizajne programu často neefektívne.

Grand Central Dispatch (GCD) je technológia vyvíjaná spoločnosťou *Apple Inc.* pre optimalizáciu využívania ich viac-jadrových zariadení. Tento framework implementovaný v jazyku C s rozhraniami v jazykoch Swift a Objective-C má podporu operačných systémov konkrétnych zariadení. GCD poskytuje programátorom rozhranie, pri ktorom nemusia manažovať jednotlivé vlákna. Tým uľahčuje paralelizmus pre programátora a robí ho efektívnym.

Cieľom tejto práce je implementovať knižnicu vo zvolenom štandarte jazyka C++, ktorá bude čo najlepšie replikovať chovanie pôvodnej implementácie. Táto knižnica nemá priamu podporu operačných systémov jednotlivých zariadení ale zároveň je prenositeľná medzi rôznymi prostrediami. Implementovaná knižnica zjednodušuje využitie vláken pre programátorov, ale kvôli jej prenositeľnosti nevyužíva možnosti zariadenia pre najefektívnejšie rozdelenie paralelnej práce.

V prvej časti sa približuje teória potrebná pre pochopenie paralelného behu progamu. Druhá kapitola popisuje ako existujúci systém Grand Central Dispatch funguje. Tretia časť približuje návrh implementovanej knižnice pri využití zvoleného štandartu jazyka C++. Posledná časť popisuje základné a najbežnejšie vzory použitia knižnice a zároveň vyhodnocuje fungovanie knižnice pri testovaní na príkladoch a zariadeniach s rôznym počtom procesorov.

Kapitola 2

Viacvláknový beh programu

V tejto kapitole sú priblížené základné koncepty potrebné pre pochopenie problematiky paralelného behu programu. Vysvetluje termíny ako súbežnost a paralelizmus, vlákno, kritická sekcia alebo threadpool, ktoré sú neskôr používané pri popise a návrhu systému Grand Central Dispatch.

Pri paralelnom resp. súbežnom behu je program rozdelený na celky, ktoré môžu byť vykonávané nezávisle. Zariadenia s viacerými jadrami procesorov môžu tieto časti vykonávat naraz. Pri rozdelení programu na väčší počet úloh ako dostupných procesorov je zabezpečený pseudo-paralelizmus (súbežnosť) využitím zmeny kontextu.

Súbežnosť a paralelizmus

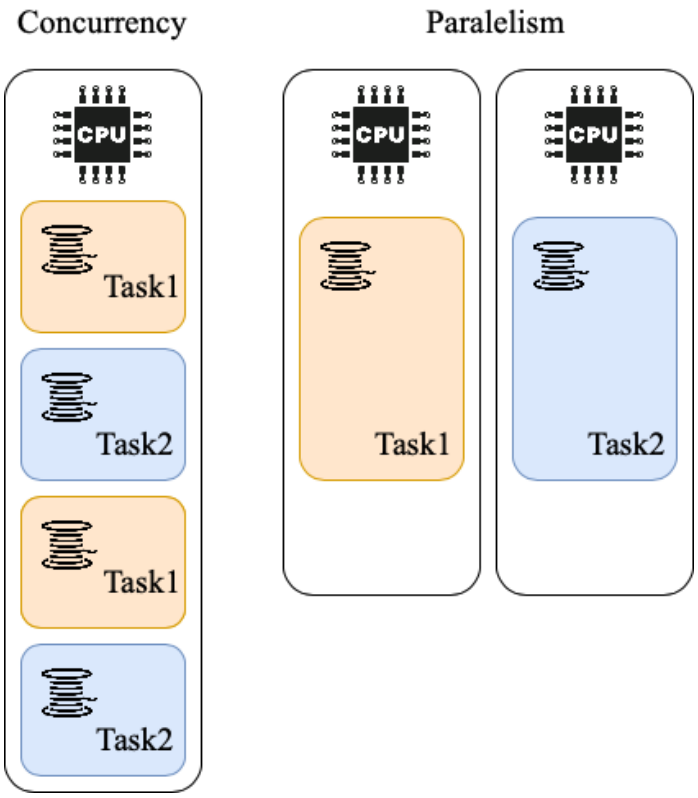
Pri operačných systémoch, ktoré su schopné vykonávať viacero úloh súbežné môže nastať situácia kedy je počet vykonávaných úloh väčší ako počet dostupných procesorov. Operačný systém zabezpečuje pseudo-paralelizmus tým, že prenecháva procesor bežiacim vlákna v stanovených intervaloch (angl. time-slicing). Je dôležité ustanoviť pojmy paralelizmus a súbežnosť:

- **Súbežnosť** - program pracuje na viac úlohách naraz. Pri väčšom počte úloh ako dostupných procesoroch nastáva zmena kontextu.
- **Paralelizmus** - jedna úloha je rozdelená na nezávislé celky, ktoré sú vykonávané na dostupných procesoroch.

Zmena kontextu

Operačný systém pri vykonávaní viacerých úloh pseudo-paralelne využíva operáciu zmeny kontextu. Zmena kontextu uloží stav bežiaceho procesu alebo vlákna, pozastaví ho a prenechá procesor dalšiemu procesu alebo vláknu s jeho kontextom.

Zmena kontextu je výpočetne drahá operácia a preto je v programe potrebné udržovať nízky počet vláken, ktoré plne využívajú procesor bez čakania na uvoľnenie zámku alebo dokončenie I/O operácie.



Obr. 2.1: Súbežnosť vs paralelizmus

Vlákno

Použitím vláken je program resp. proces schopný rozdeliť svoj beh na paralelne, prípadne pseudo-paralelne, bežiace úlohy. Vlákna bežia nezávisle od ostatných vláken. Sú obsiahnuté v jednom procese a všetky zdielajú zdroje ako napríklad pamäť procesu.

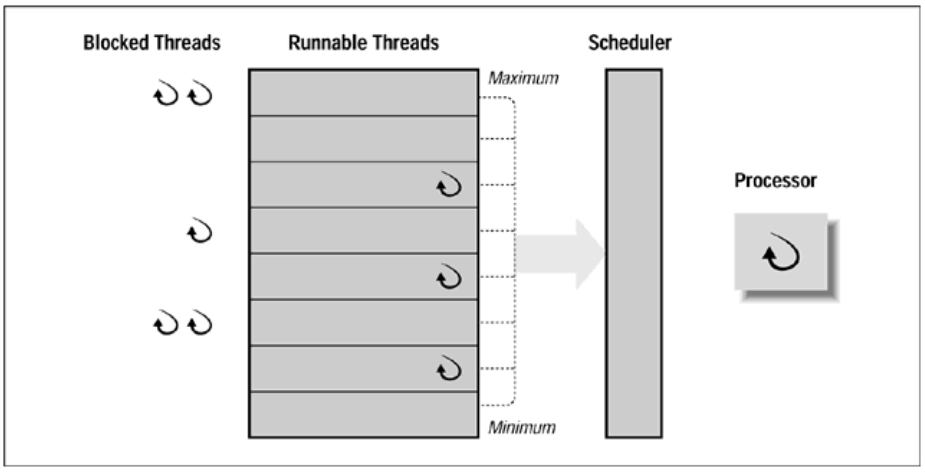
Kvôli rôznorodosti operačných systémov nie je implementácia vláken rovnáka. Preto je definovaný štandart **Pthreads**. Je to rozhranie nad vláknami a synchronizačnými prvkami UNIX systémov. Vychádza z prenosného rozhrania POSIX (Portable Operating Interface), ktorého štandard určuje ako majú POSIX-kompatibilné systémy fungovať [6].

Plánovač

Ak v je v systéme viacero vláken je potrebné, aby sa vybrali tie, ktoré majú bežať. Plánovač je súčasť operačného systému, ktorá riadi ako vlákna zdielajú procesor. Štandard pthreads tiež definuje rozhranie pre stanovanie správania vláken vzhľadom na plánovač . Poskytuje nastavenie dvoch vlastností - **Scheduling priority** a **Scheduling policy**.

Scheduling priority

Priorita je vlastnosť určujúca, ktoré vlákno ma preferenciu pri využití procesora vzhľadom na iné vlákna. Plánovač prehľadáva pole priorít, ktoré obsahuje frontu vláken pre každú prioritu, ako je možné vidieť na obrázku 2.2. Z tohto pola vyberá vlákno s najvyššou prioritou, ktoré je schopné behu, t.j. nečaká na I/O operáciu alebo na uvoľnenie zámku.



Obr. 2.2: Pole priorít vláken pre plánovač [1]

Vždy keď je možné vykonávať vlákno s vyššou prioritou ako vlákno, ktoré práve využíva procesor nastane prerušenie behu tohto vlákna. Nastáva nedobrovolná zmena kontextu a vlákno s vyššou prioritou získa prístup k procesoru.

Scheduling policy

Je to spôsob ako vlákna s rovnakou prioritou dostávajú prístup k procesoru. Existujú 3 hlavné spôsoby:

- **SCHED_FIFO** - *first-in first-out* spôsob vykonávania vlákna prenecháva procesor vláknu s najvyššou prioritou až do konca jeho vykonávania alebo kým nie je blokované. Pri zlom využití tohto spôsobu plánovania je možné vyhladovať ostatné vlákna s rovnakou alebo nižšou prioritou.
- **SCHED_RR** - tento spôsob pridáva FIFO spôsobu časový blok pre vlákno, po ktorom sa musí vzdať miesta na procesore.
- **SCHED_OTHER** - štandardný spôsob pri vytvorení nového vlákna. Jeho implementácia sa líši medzi systémami, ale základom zostáva najnižšia a fixná priorita vzhladom k ostatným spôsobom plánovania. Vlákna plánované týmto spôsobom majú tiež stanovený časový blok, po ktorom sa musia vzdať procesora.

Na Linux systémoch je dostupný ďalši spôsob - **SCHED_IDLE**. Tento spôsob je pre dlhé operácie na pozadí, ktoré by mali mať najnižšiu prioritu v systéme.

Je dôležité podotknúť, že existuje možnosť zmeniť prioritu aj pri planovaní spôsobom SCHED_OTHER. Táto priorita sa ale vzťahuje na celý proces a preto sa tzv. *nice value* v implementovanej knižnici nepoužíva. Tento spôsob zmeny priority je využitý iba pri operačnom systéme macOS. Ten neimplementuje spôsob SCHED_IDLE ale ponúka nastavenie priority vlákna na **PRIO_DARWIN_BG** pri spôsobe SCHED_OTHER.

[1] Zdroj: http://maxim.int.ru/bookshelf/PthreadsProgram/htm/r_37.html

Kritická sekcia

Zdieľanie zdrojov a nezávislosť behu vláken vytvára situáciu, kedy môžu nastať neočakávané problémy alebo nedeterministické chovanie programu. Kritická sekcia je časť programu, kedy vlákna môžu súbežne pristúpiť k zdielanému zdroju ako napríklad dátová štruktúra. Kvôli tomu treba zabezpečiť jednotlivý prístup k zdroju. Pre riadenie a synchronizáciu vláken existujú synchronizačné prostriedky.

Mutex

Mutual-exclusion je synchronizačný prostriedok, ktorý zabezpečuje, aby viac vláken nepristupovalo ku kritickej sekcii. Mutex má jeden z dvoch stavov: volný alebo vlastnený. Ak je mutex vlastnený procesom alebo vláknom, nemôže do kritickej sekcie vstúpiť ďalší. Mutexy sú využité pri realizácii viacerých synchronizačných primitív ako napríklad zámky alebo semafóry.

Threadpool

Threadpool je návrhový vzor, ktorý je odvodený od návrhového vzoru Object Pool. Tento vzor využíva predom inicializovanú množinu objektov, ktoré následne opakovane využíva namiesto toho, aby sa pri každom použití objekt vytváral a odstraňoval. Tento vzor je využitý hlavne pre zvýšenie výkonu. Keďže vytváranie a odstraňovanie vláken je náročná operácia je tento návrhový vzor využitý práve pri programoch, ktoré potrebujú mať vlákna opakovane dostupné. Vláknam sú následne odovzdávané úlohy, ktoré majú vykonať. Porovnanie výkonu vzoru threadpool a vytvárania vláken je priblížené v sekcii 5.1.

Dôležitou vlastnosťou vzoru Object Pool je počet objektov, ktoré sa v množine nachádzajú. Pri určovaní optimálneho počtu vláken v množine je potrebné zachovať najvyššie možné výpočetné vyťaženie. Pri predpoklade, že vykonávané úlohy nie sú blokujúce, t.j. nečakajú na I/O operáciu alebo uvoľnenie zámku, je optimálny počet vláken rovný počtu jadier resp. procesorov dostupných na zariadení. Pri tomto počte je eliminovaná nadbytočná zmena kontextu a zároveň sú vyťažené všetky procesory.

Kapitola 3

Grand Central Dispatch

Grand Central Dispatch [2] je súborom vlastností implementačného jazyka, run-time knižníc a vlastností operačných systémov, ktorý dopomáha rýchlejšiemu a efektívnejšiemu behu kódu na viac-jadrových zariadeniach macOS, iOS, watchOS a tvOS. Bližšie informácie o internom fungovaní systému je možne nájsť v konferenciách od firmy Apple, napr. [3].

Pri využití GCD programátor nie je vystavený spravovaniu jednotlivých vláken. Namiesto toho sa používa rozhranie tzv. *Dispatch Queue*. Sú to fronty do ktorých sa zaraďujú jednotlivé úlohy, ktoré treba vykonať. Tieto úlohy sú zapúzdrené do objektu *Dispatch Workitem*, ktorý poskytuje niekoľko metód a atribútov na synchronizáciu a riadenie. Fronty klasickým *First-In First-Out* spôsobom bloky ďalej odovzdávajú vláknam, ktoré sú v GCD udržiavané využitím návrhového vzoru *threadpool*. Fronty Dispatch Queue sú hlavnou entitou pre riadenie paralelizácie a synchronizácie týchto úloh.

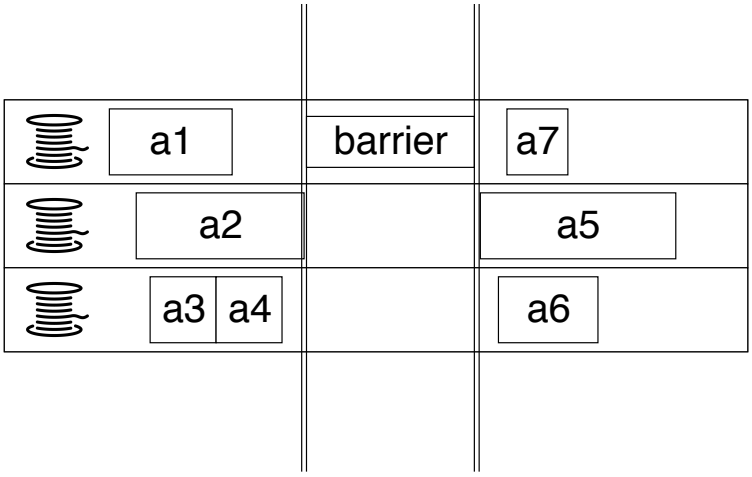
Grand Central Dispatch poskytuje aj iné možnosti synchronizácie pomocou *Dispatch Semaphore* a *Dispatch Group*. Bloky je taktiež možné priradiť k zdroju určitej systémovej udalosti použitím *Dispatch Source*. Ak daná udalosť nastane, úloha sa pridá do zvolenej fronty. Základné objekty Grand Central Dispatch teda sú:

- **Dispatch Workitem** - je objekt konkrétnej úlohy. Úloha je kus alebo blok kódu, ktorý je možné samostatne vykonať a zachytáva aj rámec mimo toho svojho. Po zapúzdrení tohoto bloku do objektu sa s ním ďalej pracuje vo frontách na zabezpečenie paralelizácie a synchronizácie ich vykonania.
- **Dispatch Queue** - je najzákladnejšia entita, ktorá spája bloky a vykonávajúce vlákna. Knižnica ponúka fronty s viacerými prioritami vykonania - Quality of Service. Podľa týchto priorít a možností systému sa rozhodne v akom množstve sa úlohy vykonávajú. Existujú dva druhy front. *Serial* je typ ktorý zabezpečuje, že sa vykonáva práve jeden blok pridaný do tejto fronty. Vo fronte *concurrent* sa bloky začnú vykonávať v poradí v ktorom boli pridané, ale môže sa ich vykonávať viac naraz.
- **Dispatch Source** - je objekt, ktorý do front pridáva zvolený blok ak nastane systémova udalosť. Tieto udalosti zahŕňaju napríklad časovač, POSIX signály alebo či je socket pripravený na I/O operácie.
- **Dispatch Group** - umožňuje združovanie blokov do skupín. Toto zoskupovanie funguje na synchronizáciu, kedy je možné čakať kým sa všetky bloky vykonali.
- **Dispatch Semaphore** - sa správa ako počítajúci semafór. Dovoluje vykonanie len určitého počtu blokov súbežne.

3.1 Dispatch Workitem

V Grand Central Dispatch sa pri paralelnom vykonávaní kódu používajú tzv. úlohy alebo bloky. Blok kódu je samostatne spustitelný a od funkcií sa líši tým, že dokáže zachytiť premenné aj mimo svoj rámec. Tento blok je reprezentovaný objektom Dispatch Workitem. Vďaka tomu je možne k bloku pridať dalšie bloky ako fallback, ktoré sa pridajú do front po jeho vykonaní, rôzne blok konfigurovať pomocou flagov, zrušiť ho ešte predtým ako bol vykonaný alebo blokovať vykonávanie vlákna kým sa blok nevykoná.

Špeciálny typ bloku, ktorý slúži na synchronizáciu na frontách je bariérový blok. Na obrázku 3.1 je vidieť ako tento blok funguje. Ak je bariérový blok na vrchole fronty, threadpool nedovolí vláknam tento blok z fronty odstrániť a začať vykonávať kým sú vykonávané iné bloky, ktoré vlákna dostali z tejto fronty. Tento blok je možné začať vykonávať až keď skončí vykonávanie týchto blokov. Počas vykonávania bariérového bloku je fronta taktiež odstavená až dovtedy, kým sa vykonávanie bloku nedokončí.

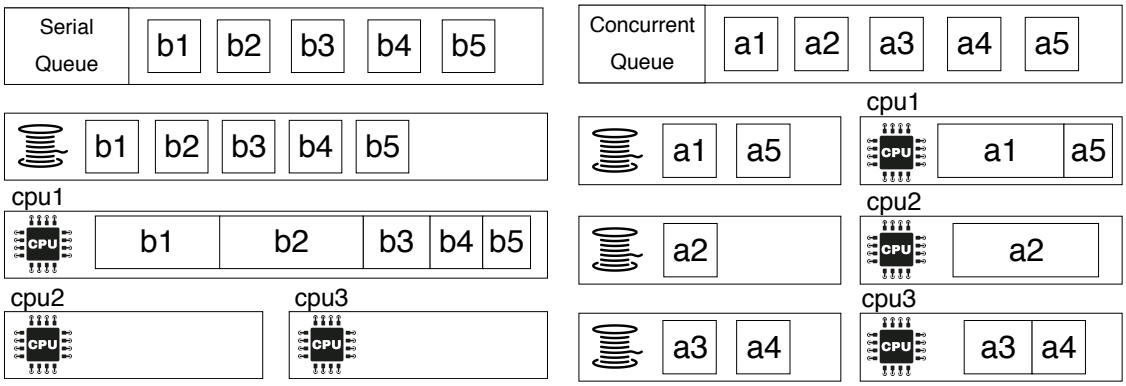


Obr. 3.1: Bariérový blok vo fronte typu concurrent

3.2 Dispatch Queue

Dispatch Queue je základnou entitou GCD. Spolupracuje s objektom threadpool pri vykonávaní blokov. Uľahčuje programátorovi paralelizáciu tým, že nemusí využívať rozhranie na úrovni vláken. Všetky fronty sú dátové štruktúry typu *FIFO*. To znamená, že úlohy sa začnú vykonávať v takom poradí v akom boli do nej pridané.

Dispatch Queue má dva základné typy a to *serial* a *concurrent*. Fronta typu serial po vydaní bloku vláknam uvolní ďalší až keď sa vykonávanie tohto bloku dokončí. Fronta typu concurrent po vydaní bloku nečaká na jeho dokončenie. Postupne vydáva bloky na všetky dostupné vlákna. To znamená, že sa môže vykonávať viacej blokov z jednej fronty paralelne naraz. V concurrent fronte môže nastať situácia, kedy postupnosť vloženia blokov do fronty neodpovedá postupnosti dokončenia ich vykonávania. Obrázok 3.2 ukazuje rozdiel vykonávania blokov pri týchto typoch.



(a) Serial fronta

(b) Concurrent fronta

Obr. 3.2: Rozdiel medzi serial a concurrent frontou pri troch procesoroch

Bloky je možné do front pridávať dvomi spôsobmi. Synchrónne pridanie úlohy do fronty blokuje vlákno, ktoré úlohu odovzdalo, kým nie je vykonaná. Asynchrónne pridávanie blokov je častejšie a zabezpečuje, že úloha je vykonaná paralelne zatiaľ čo vlákno, ktoré úlohu odovzdalo pokračuje vo vykonávaní kódu ďalej. Špeciálny typ asynchrónneho pridania naplánuje odovzdanie úlohy fronte po ubehnutí zvoleného času.

```
DispatchQueue.main.async {
    print("Hello")
}
DispatchQueue.global().async {
   // otevri soubor
   DispatchQueue.main.sync {
      // posli zpravu UI, soubor otevren
   }
}
```

Výpis 3.1: Využitie front v jazyku Swift

Pri programovaní aplikácií systém ponúka globálne fronty. Pre každú hodnotu Quality of Service, ktoré su bližsie popísané v sekcii 3.5, je vytvorená globálna fronta typu concurrent. Odporúča sa využívať tieto globálne fronty, pretože pri vytvorení súkromných Dispatch Queue sú alokované nové vlákna pre aplikáciu. To môže viesť k nadmernému počtu alokovaných vláken. Jedna globálna fronta typu serial je tiež vytvorená pre hlavné vlákno aplikácie, ktoré má najčastejšie na starosti elementy grafického rozhrania.

Thread explosion

Pri využití Grand Central Dispatch môže nastať situácia, kedy sa vytvorí príliš veľa vlákien. Toto sa nazýva *thread explosion*. Pri vytvorení nadbytočného počtu vláken prichádza k zníženiu efektivity, pretože procesor musí vykonávať operáciu zmeny kontextu, namiesto toho, aby vykonával bloky. Nadbytočný počet alokovaných vláken môže nastať v dvoch prípadoch.

- Threadpool alokuje nové vlákna pre každú novú vytvorenú súkromnú Dispatch Queue. Kvôli tomu môže byť vytvorených viac vláken ako je potreba. Preto nastáva problém s nadmernou zmenou kontextu pri vykonávaní blokov z viacerých vláken.
- Úlohy pridané do concurrent fronty môžu byť blokujúce. Pri vyskytnutí takéhoto bloku GCD udržiava rovnaký počet pracujúcich vláken, a preto threadpool alokuje nové.

Medzi blokujúce úlohy patrí uspanie vlákna, čakanie na I/O operáciu alebo čakanie na uvoľnenie zámku.

3.3 Dispatch Source

Programy využívajúce Grand Central Dispatch sú najčastejšie aplikácie s architektúrou udalosťami riadeného behu. Dispatch Source je objekt, ktorý slúži na monitorovanie špecifických systémových udalostí. Ak zvolená udalosť nastane, Dispatch Source naplánuje blok na vykonanie na zvolenej fronte. Typy udalostí, ktoré sú monitorované:

- **Timer** - periodicky generujúce udalosti. Po ubehnutí zvoleného intervalu pridá blok do zvolenej fronty.
- **POSIX signal** - upozorňuje, ak nastal POSIX signál.
- **Descriptor** - upozorňuje pri súborových a socketových operáciách. Keď je možné čítať alebo vpisovať alebo keď je súbor zmazaný, presunutý, premenovaný alebo má zmenené metadáta.
- **Process** - upozorňuje pri udalostiach týkajúcich sa systémových procesov ako napr. keď je proces ukončený, dostane signál alebo použije `fork` a `exec`.
- **Mach port** - upozorňuje pri Mach udalostiach. Mach je kernel vyvinutý pre podporu operačných systémov využívajúcich paralelného vykonávania na viacerých procesoroch. Operačne systémy Apple zariadení sú od neho odvodené.

Dispatch Source nahradzuje implementovanie callback funkcií pri monitorovaní systémových udalostí a umožnuje vytváranie úloh pre fronty narozdiel od manuálneho pridávania. Každa monitorovaná udalosť pridá do front rovnakú úlohu priradenú k objektu Dispatch Source.

3.4 Synchronizácia vykonania úloh

Pri paralelnom programovaní nastávajú situácie, kedy potrebujeme úlohy (v prípade GCD bloky kódu) synchronizovať. Buď je potrebné obmedziť paralelný prístup k zdielanému zdroju alebo chceme vedieť, kedy bola vykonaná určitá skupina blokov. Na toto sú v Grand Central Dispatch využité dve entity - Dispatch Semaphore a Dispatch Group. Príklad ich použitia je možno vidieť na výpise 5.4 a 5.5.

3.4.1 Dispatch Semaphore

Trieda Dispatch Semaphore je jednoduchý počítajúci semafór. Vo všeobecnosti je semafór abstraktný dátový typ, ktorý sa používa na kontrolu prístupu k zdielanému zdroju z

viacerých procesov alebo vláken pri paralelnom behu. Semafór môže byť binárny alebo počítajúci. Binárny semafór môže nadobúdať len hodnoty 0 a 1 (zamknutý/odomknutý). Často sa binárny semafór využíva pri implementácii zámkov.

Počítajúci semafór dovoľuje prístup z viacerých vláken, v prípade GCD blokov, na základe predom určenej hodnoty. Vlákno pri nenulovej hodnote semafór dekrementuje a pri odchode z kritickej sekcie semafór inkrementuje. Pri nulovej hodnote vlákno čaká, kým kritickú sekciu opustí iné vlákno. Najtypickejším prípadom užitia počítajúceho semafóru je obmedzenie prístupu k zdielanému zdroju pre ľubovolný počet vláken naraz.

3.4.2 Dispatch Group

Dispatch Group je ďalšia možnosť synchronizácie úloh v GCD. Funguje podobne ako Dispatch Semaphore. Jej využitie je však iné. Dispatch Group sa využíva na združenie viacerých Dispatch Workitem objektov. Dispatch Group sa neinicializuje s počiatočnou hodnotou a metóda, ktorá dekrementuje počítadlo neblokuje vykonávanie vlákna. Po dokončení všetkých blokov, ktoré vošli do tejto skupiny je počítadlo rovné hodnote 0. Keď je počítadlo rovné 0 sú uvolnené všetky vlákna, ktoré na dokončenie skupiny čakali. Taktiež je možné naplánovať pridanie blokov do fronty po vykonaní celej skupiny.

3.5 Quality of Service

Jeden z dôvodov prečo bolo GCD vytvorené je programovanie užívateľských aplikácií a práve preto je dôležité, aby bola aplikácia responzívna pre úživatela aj keď vykonáva rôzne procesy v pozadí ako napríklad sťahovanie dát, náročný výpočet alebo prístup do databázy.

Quality of Service určuje ako dôležítá je úloha, ktorú treba vykonať. Na základe zvolenej hodnoty QoS systém rozhodne koľko vláken použije a kedy prácu vykoná v súlade s možnosťami systému ako napríklad zaťaženie procesora, teplota zariadenia alebo stav batérie. Tým sa stáva program efektívnejší z pohľadu zariadenia na ktorom beží.

Quality of Service je koncept, ktorý využíva plánovač konkrétnych operačných systémov. V týchto operačných systémoch je možné pomocou QoS nastaviť prioritu vláknam, frontám aj blokom. Je možné nastaviť 4 hlavné stupne:

- **User Interactive** - úlohy interaktívne s užívatelom ako napr. animácie, vykreslovanie grafického rozhrania alebo spracovanie udalostí užívatelského vstupu.
- **User Initiated** - úlohy ktoré sú potrebné pre funkcionalitu programu ako napr. načítanie dát, ktoré sa majú zobraziť.
- **Utility** - úlohy ktoré sa vykonávajú dlhšiu dobu a nie sú potrebné pre interkaciu s programom ako napr. sťahovanie dát.
- **Background** - úlohy o ktorých užívatel nevie a vykonávajú sa na pozadí ako napr. zálohovanie alebo synchronizácia.

Existujú ešte stupne *default* a *unspecified*. Default sa nachádza medzi User Initiated a Utility. Túto hodnotu majú fronty, ktoré nemajú hodnotu QoS nastavenú.

Kapitola 4

Návrh a implementácia systému v C++

Úlohou bolo implementovať Grand Central Dispatch ako knižnicu využitím zvoleného štandardu jazyka C++. Táto knižnica by mala čo najlepšie replikovať správanie GCD. Narozdiel od pôvodnej implementácie, ktorá má podporu zabudovanú priamo v operačných systémoch jednotlivých zariadení bude táto knižnica prenositelná medzi rôznymi platformami. Rozhranie knižnice pre programátorov sa podobá tomu v jazyku Swift s potenciálnymi zmenami kvôli odlišnosti jazykov.

Pri implementácii bol zvolený štandart C++17. V dobe implementácie najnovší dostupný štandard, ktorý zároveň podporuje prenositelnú podporu vláken `std::thread` a potrebné *lambda expressions* zavedené v C++11. Základný návrh efektívneho využívania rozhrania vláken v C++ bol inšpirovaný knihou [4] a open-source portom Grand Central Dispatch [1]. Taktiež pri implementácii knižnice boli využité manualové stránky Linuxu[5] a jazyka dokumentácia jazyka C++ [1].

4.1 Abstraktné triedy

Abstraktné triedy sú triedy, ktoré nie sú inštanciované a fungujú ako rozhrania, ktoré ostatné objekty implemenujú. Pri návrhu základných entít, konkrétne front, blokov, semafórov a skupín, sa prekrývali niektoré ich vlastnosti. Abstraktné triedy `Dispatch::Work` a `Dispatch::Object` implementujú zdielané metódy a rozhrania.

Dispatch::Work

Implementuje rozhranie čakania na dokončenie a odovzdávanie callback blokov do front. Toto rozhranie je implementované triedami Dispatch::Group a Dispatch::Workitem. Trieda Dispatch::Semaphore implementuje iba rozhranie čakania. Využitím rozhraní `std::mutex` a `std::condition_variable` ovláda zámok, ktorý zastaví vykonávanie vlákna. Toto rozhranie v odvodených triedach je možné ovladať metódami:

- `void wait()` - vlákno čaká kým je blok alebo skupina vykonaná

[1] https://github.com/apple/swift-corelibs-libdispatch

- `std::cv_status wait(uint32_t miliseconds)` - vlákno podobne čaká ako pri predošlej metóde ale maximálne zvolený čas. Návratová hodnota metódy je určená tým, či bol zámok odomknutý alebo ubehol stanovený čas.
- `void notify(Dispatch::Queue *q, Dispatch::Workitem *wi)` - pridá callback bloky do fronty po vykonaní práce.

Dispatch::Object

Táto abstraktná trieda pridáva blokom a frontám počítadlo referencií. Keďže tieto entity sú často navzájom referencované, je potrebné ich uvolniť až vtedy, keď už reálne žiadnu referenciu nemajú. Metódy `void retain()` a `void release()` inkrementujú a dekrementujú počítadlo. Ďalšie rozhranie je pozastavenie vydávania blokov pre fronty a zdroj udalostí. Preto je implementované ďalšie počítadlo pozastavení, ktoré je ovladané metódami `void suspend()` a `void resume()`.

4.2 Rozhranie front a blokov

Lambda výrazy

Ako bolo vyššie spomenuté Dispatch Workitem je zapúzdrenie samostatne spustielného kódu do objektu. Tento blok je v knižnici implementovaný využitím tzv. *lambda expressions*, ktoré su obdoba *closures* jazyka Swift. Lambda výrazy sú anonymné funkcie často používane na krátke kusy kódu, ktoré nemá zmysel pomenovávať alebo znova používať. Od funkcií sa líšia tým, že vedia zachytiť premenné, ktoré sa nachádzajú v rámci ich definície. Syntax lambda výrazu obsahuje pole *captures*, v ktorom sa definuje či má blok zachytiť premennú referenciou alebo hodnotou. Existujú tri možnosti definície:

- [&] - zachytí všetky premenné podľa referencie
- [=] - zachytí všetky premenné podľa ich hodnoty
- [a, &b] - zmiešaná defínicia, pri ktorej je možné určit typ zachytenia pre každú premennú.

Bloky

Aby k bloku bolo možné pridávať atribúty a metódy je potrebné ho zapúzdriť do objektu. Objekt Dispatch::Workitem je možné vytvoriť iba alokovaním na halde kvôli internému počítadlu referencií z abstraktnej triedy Dispatch::Object. To znamená, že je možné ho inicializovať iba operátorom `new`. Tento objekt sa vytvára aj interne pri funkciách knižnice, ktorých argument je iba lambda výraz.

Pri vytváraní objektu je možné mu nastaviť skupinu prepínačov, ktoré su reprezentované štruktúrou `Dispatch::workitem_flags`. Obsahuje tri prepínače dátového typu bool:

- `release_after_completition` - rozhoduje, či má byť počítadlo referencií znížené po vykonaní bloku. V podstate nahradzuje operátor `delete`.
- `barrier` - rozhoduje, či je blok bariérový. Funkcia bariérového bloku bola ukázaná na obrázku 3.1.

- `enforce_qos` - rozhoduje, či má blok zdediť prioritu vykonania z fronty v ktorej sa nachádza alebo modifikovať správanie fronty na základe priority bloku.

Ako je zjávné z prepínaču enforce_qos, blok môže mať vlastnú prioritu vykonania. Tú je možné nastaviť metódou `void setQoS(Dispatch::qos qos)`. Blok je taktiež možné zrušiť metódou `void cancel()`. Po zrušení blok stále zostáva vo fronte, pri prepínači enforce_qos sa fronty správaju rovnako a vlákna ho spracúvajú rovnakým spôsobom ako nezrušené bloky. Po spracovaní sa taktiež odovzdajú callback bloky pridané metódou `notify` do fronty. Jediný rozdiel v procese vykonania bloku je, že sa nevykoná lambda výraz.

Algoritmus 5 ukazuje priebeh vykonania bloku. Rozhraním triedy Work zobudí vlákna čakajúce na dokončenie bloku, zobudí vlákno, ktoré zavolalo metódu synchrónneho pridania do fronty, odozvdá frontám bloky pridané metódou `notify` a pri nastavenom flagu zníži svoj počet referencií.

Algoritmus 1: Vykonanie bloku

```
if not this.canceled then
    execute();
end
submit_callback_blocks();
Work::condition_variable.notify_all();
Queue::sync_condition_variable.notify_one();
if flags.release_after_completion then
    Object::reduce_references();
end
```

Fronty

Tak isto ako Dispatch::Workitem je zapúzdrenie lambda výrazu do objektu kvôli pridanej funkcionalite, je `std::queue<Dispatch::Workitem *>` zapúzdrená do objektu Dispatch::Queue. Hoci je možné vytvoriť si vlastné súkromné fronty, tak knižnica poskytuje funkcie:

- `Dispatch::Queue *get_main_queue()` - navráti globálnu frontu typu serial. Táto fronta reprezentuje hlavné vlákno.
- `Dispatch::Queue *get_global_queue(Dispatch::qos qos)` - navráti globálnu frontu typu concurrent podľa zvolenej priority.

Tieto globálne fronty sú vytvorené spolu so singletonom threadpool. Rozhranie fronty zostáva rovnaké a je možné blok pridať štyrmi spôsobmi:

- `void async(Dispatch::Workitem* block)` - asynchrónne pridanie bloku. Blok sa pridá do fronty a vlákno pokračuje vo vykonávaní kódu.
- `void sync(Dispatch::Workitem* block)` - synchrónne pridanie bloku. Blok pridá vlákno do fronty a vlákno čaká, kým sa blok vykoná. Viz. algoritmus 2.
- `void async_after(Dispatch::Workitem* block, uint32_t ms)` - asynchrónne načasované pridanie. Blok je asynchrónne pridaný po zvolenom čase. Viac v sekcii 4.4.

- `void concurrent_perform(u_int32_t iterations, std::function<void(int)>)` - využíva sa pri pridaní viacero rovnakých blokov.

Algoritmus 2: Pridanie bloku do fronty metódou sync

```
Input: Dispatch::Workitem workitem, Dispatch::Queue queue
workitem.sync_condition_variable = queue.sync_condition_variable;
workitem.Object::increase_references();
if workitem.flags.enforce_qos then
    if workitem.qos > queue.qos then
        queue.qos = workitem.qos;
    end
end
queue.lock();
queue.push(workitem);
queue.unlock();
if threadpool::try_wakeup_thread(queue.qos) then
    threadpool::condition_variable.notify_one();
end
queue.sync_cv.wait();
```

concurrent_perform

Spôsob pridania concurrent_perform optimalizuje pridávanie blokov v iterácii. Tento typ volania je synchrónny - volajúce vlákno pokračuje až po vykonaní všetkých blokov. Pri vykonávaní viacero blokov, ktoré nie sú veľmi náročné pre procesor môže réžia knižnice spomalovať vykonávanie pre paralelný výpočet. Táto metóda spojí viacero iterácií do jedného bloku. To znamená, že počet blokov pridaných do fronty je rovný počtu dostupných vláken v objekte threadpool. Ak je potrebné pristúpiť k indexu iterácie je možné metódu zavolať s lambda výrazom so vstupným parametrom, viz. výpis 4.1.

```
auto queue = get_global_queue(DISPATCH_QOS_DEFAULT);
queue->concurrent_perform(1000,[](int i){
   printf("iteration: %d\n",i);
});
```

Výpis 4.1: Prístup k indexu iterácii v bloku pridanom spôsobom concurrent_perform

4.3 Threadpool

Základnou entitou fungovania celého systému je objekt spravujúci vlákna. Objekt využíva návrhové vzory threadpool a singleton. Návrhový vzor threadpool je vysvetlený v kapitole 2. Návrhový vzor singleton obmedzuje počet inštancií triedy v programe na jednu. Využíva sa, keď práve jeden taký objekt je potrebný na fungovanie systému. Základná definícia vzoru singleton v jazyku C++, ktorá obmedzuje inštanciáciu triedy na jeden objekt ukazuje príklad vo výpise 4.2.

```
class Threadpool{
public:
   //singleton should not be clonable
   Threadpool(const Threadpool&) = delete;
   //singleton not be assignable
   void operator=(const Threadpool&) = delete;
   //singleton getter
   static Threadpool& getInstance();
private:
   Threadpool();
   ~Threadpool();
}

Threadpool &Threadpool::getInstance() {
   static Threadpool instance;
   return instance;
}
```

Výpis 4.2: Definícia triedy návrhového vzoru Singleton

Vlákna v objekte threadpool spracovávajú úlohy, ktoré získavajú z front. Pri využití knižnice Grand Central Dispatch sú dostupné 3 typy vláken na vykonávanie kódu:

- **Hlavné systémové vlákno** - vlákno, ktoré je implicitne dostupné pri sekvenčnom programe so vstupom vo funkcii main().
- **Hlavné GCD vlákno** - vlákno, ktoré obsluhuje hlavnú globálnu frontu typu serial pri využití architektúry riadenej udalosťami.
- **Vykonávajúce vlákna** - skupina vláken, ktoré paralelne resp. súbežne pri využití rôznych hodnôt Quality of Service, vykonávajú úlohy zo súkromných front a globálnych front typu concurrent.

4.3.1 Určenie počtu vláken v objekte threadpool

Pri vykonávajucích vláknach je dôležité určit ich počet. Implementovaný systém ma zvolený konštantný počet vykonávajúcich vláken pri inicializácii objektu threadpool, hoci v pôvodnej implementácii Grand Central Dispatch je počet vláken dynamický. Dynamický počet vláken v systéme zabezpečuje, že procesor je stále dostatočne vyťažený aj pri vykonávaní

kódu, ktorý nie je priamo naviazaný na procesor, príkladom takého kódu je napríklad čakanie na ubehnutie intervalu, I/O operácie alebo čakanie na uvoľnenie zámku. Implementovaná knižnica je navrhnutá hlavne pre vykonávanie kódu využívajúceho procesor.

Použitím `std::thread::hardware_concurrency()` je v C++ možné získať počet dostupných jadier zariadenia. Kvôli zabezpečeniu nedobrovolnej zmeny kontextu pri blokoch s vyššou prioritou je ale potrebné, aby vláken bolo viac.

Predpokladajme situáciu so zariadením s N procesormi. Počet dostupných vláken je taktiež N a všetky sú zamestnané dlhou prácou s prioritou Background. V situácii kedy by prišla práca s vyššou prioritou, by práca musela čakať na dokončenie bloku a uvoľnenie vlákna. Preto je zvolený počet vláken v objekte threadpool rovný $2N$. Fungovanie vláken vzhľadom na úroven Quality of Service je popísana v sekcii 4.3.3.

4.3.2 Run Loop

Ako už bolo spomenuté vyššie, vlákna sú vytvorené iba pri inicializácii objektu threadpool. To eliminuje réžiu, ktorá by vznikala pri vytvorení osobitného vlákna pre každý Dispatch::Workitem. Pri vytvorení vlákna je potrebné zvoliť funkciu, ktorú bude vykonávať. Po jej skončení sa vlákno odstráni. Kvôli tomu je potrebné vytvoriť tzv. *run loop*. Je to funkcia, ktorá definuje priebeh spracovania bloku v nekonečnom cykle. V implementovanej knižnici je možné nájst 3 typy run loop funkcií.

- **Run loop hlavného vlákna GCD** - implementuje logiku hlavnej fronty typu serial a spracovanie bloku. Viz. algoritmus 3.
- **Run loop vykonávajúcich vláken** - implementuje všetku logiku
- **Run loop časového kalendára udalostí** - vlákno, ktoré plánuje bloky podľa načasovaných udalostí. Toto vlákno je bližšie popísane v sekcii 4.4.

Run loop funkcie sú najdôležitejšie funkcie v objekte threadpool. Implementujú logiku front a udalostí. Ich priebeh je popísaný na nasledujúcich algoritmoch. Algoritmy sú písané pseudokódom namiesto jazyka C++ pre zjednodušenie.

Algoritmus 3: Run Loop hlavnej fronty

```
set_thread_priority(DISPATCH_QOS_USER_INTERACTIVE);
while true do
    condition_variable.wait();
    while true do
        main_queue.lock();
        if not main_queue.empty and main_queue.active then
            workitem = main_queue.pop();
            main_queue.unlock();
            workitem.execute();
        else
            main_queue.unlock();
            break;
        end
    end
end
```

Pri hlavnej fronte je run loop funkcia pomerne jednoduchá, pri vytvorení vlákna sa nastaví jeho priorita hodnotou Quality of Service. Ďalej iba pracuje s jednou frontou, ktorá je uzamykaná zámkom kvôli prístupu z viacerých vláken.

Algoritmus 4: Run Loop pracujúcich vláken

```
while true do
    condition_variable.wait();
    qos = DISPATCH_QOS_MAX;
    while qos >= DISPATCH_QOS_MIN do
        if can_work(qos) then
            if execute_block(qos) then
                qos = DISPATCH_QOS_MAX+1;
            end
        end
        qos–;
    end
end
```

Run loop funkcia vykonávajúcich vláken implementuje cyklus, ktorý slúži pre vyberanie blokov podľa ich priority vykonania. To zabezpečuje, že bloky s vyššou prioritou sa začnú vykonávať skôr ako bloky s nižšou prioritou.

`std::condition_variable` zabezpečuje, aby vlákno nemuselo aktívne čakať. Pri aktívnom čakaní na podmienku vlákno stále využíva procesor. Condition variable udržuje vlastnú frontu vláken, ktoré čakajú na uvoľnenie. Iné vlákno môže signalizovať uvoľnenie jedného alebo všetkých vláken čakajúcich na uvoľnenie. Vlákna čakajúce na signál nevyužívajú procesor.

Algoritmus 5 popisuje funkciu **execute_block()**, ktorá vykonáva logiku vytiahnutia bloku z fronty a jeho vykonania.

Algoritmus 5: Popis funkcie execute_block(qos) použitej v pracujúcich vláknach

```
Input  : qos ∈ <DISPATCH_QOS_MIN, DISPATCH_QOS_MAX>
Output: bool
i = 0;
while i < queues.length do
    if queues[i].qos == qos then
        queue = queues[i];
        workitem = queue.top;
        queue.lock();
        if not queue.empty then
            if queue.active then
                if queue.concurrent or queue.tasks_executing == 0 then
                    if not workitem.is_barrier or queue.tasks_executing == 0 then
                        if queue.barrier_executing then
                            set_thread_priority(qos);
                            queue.pop();
                            workitem.Object::reduce_references();
                            queue.tasks_executing++;
                            if workitem.is_barrier then
                                queue.barrier_executing = true;
                            end
                            queue.unlock();
                            workitem.execute();
                            queue.barrier_executing = false;
                            q->blocks_executed–;
                            return True;
                        end
                    end
                end
            end
        end
        queue.unlock();
    end
    i++;
end
return False;
```

Prehľadaním všetkých dostupných Dispatch::Queue v systéme funkcia vyhodnocuje, či je blok na vrchole vhodný na vykonanie. Pri nájdení takého bloku je fronta, z ktorej bol blok vytiahnutý, modifikovaná. Je inkrementovaný počet vykonávaných blokov a tak isto zmenený prepínač, ak je vykonávaný blok bariéra.

Ak sa vyhovujúci blok pre danú hodnotu Quality of Service nenájde, funkcia sa navráti a run loop vyhľadáva blok s nižšou prioritou.

4.3.3 Priorita vykonania úloh

Jednou z hlavných vlastností systému Grand Central Dispatch je určovanie priority blokov hodnotou Quality of Service. Systém na základe stavu zariadenia ako napríklad teplota, batéria alebo využitie procesorov, rozhodne, koľko vláken a koľko času venuje blokom s konkrétnou hodnotou.

Implementovaná knižnica nerozhoduje o tom, koľko procesorového času bloky dostanú vzhľadom na stav systému. Napriek tomu možnosť určenia Quality of Service stále existuje. Ako bolo spomenuté v kapitole 2, rozhranie Pthreads ponúka možnost nastavenia priority vláken. Rozhranie Quality of Service bolo pozmenené vzhľadom na možnosti rozhrania pthreds nasledovne:

- **Background** - úlohy na pozadí pri využití spôsobu plánovania SCHED_IDLE, poprípade PRIO_DARWIN_BG pri systéme macOS.
- **Utility** - úlohy so štandardnou prioritou a spôsobom plánovania SCHED_OTHER.
- **Default** - úlohy tak isto nemajú zmenenú prioritu ani spôsob plánovania, ale vlákna ich uprednostňujú pri vyťahovaní z front.
- **User Initiated** - úlohy s vyššou prioritou ako štandardné úlohy. Spôsob plánovania SCHED_RR s nastavenou najnižšou možnou prioritou.
- **User Interactive** - úlohy s najvyššou prioritou, ktorá je väčšia ako pri User Initiated. Zodpovedá priorite hlavnej fronty. Spôsob plánovania zostáva SCHED_RR.

Ako bolo spomenuté vyššie, počet dostupných vláken je $2N$. Hoci najefektívnejší počet vláken by bol N, kde N zodpovedá počtu procesorov, je dôležíté zabezpečiť, že dlhé úlohy v pozadi neblokujú novo-pridané bloky s vyššou prioritou.

Vykonávanie rovnakých úloh je stále obmedzené na počet N. Implementovaný mechanizmus po pridaní bloku do fronty rozhodne, či má zobudiť nové vlákno. Threadpool si udržiava počet bežiacich úloh a ich prioritu vykonania. Ak by malo byť vlákno zobudené s nižšou alebo rovnakou prioritou ako práve bežiacich N úloh, vlákno sa nezobudí.

Doležité je poukázať na rozdiely v prioritách. Pri hodnotách Utility a Default majú vlákna stále rovnaký spôsob plánovania. To znamená, že medzi nimi prichádza k zmene konktextu počas vykonávania a plánovač im prideluje rovnaký procesorový čas. Rozdielna je situácia pri ostatných hodnotách. Zmenou spôsobu plánovania nastáva vynútená zmena kontextu a vlákna s vyššou prioritou dostanú viac procesorového času, resp. menej pri hodnote Background.

Vyhodnotenie dopadu Quality of Service na vykonanie blokov je v kapitole 5. Na príklade je ukázná vynútená zmena kontextu pri vyšších prioritách.

4.4 Kalendár načasovaných udalostí

Systém Grand Central Dispatch sa využíva pri programoch, ktoré su riadené udalosťami. Narozdiel od sekvenčného behu program čaká na udalosti, podľa ktorých mení stav a vykonáva funkcie. V takýchto programoch je potrebné udalosti spracovávať asynchrónne, aby program nebol blokovaný pri čakaní na udalosť. Grand Central Dispatch preto ponúka rozhranie Dispatch Source.

Pri implementácii knižnice bol zvolený iba jeden typ Dispatch Source - Timer. Slúži ako ukážka asynchrónneho pridávania blokov do front pri udalosti. V tomto prípade je to časový interval. Knižnica pri inicializácii objektu threadpool vytvorí ďalšie vlákno pre asynchrónne načasované pridanie bloku do fronty. Okrem objektu `Dispatch::SourceTimer`, viz. výpis 4.3, je toto vlákno a jeho run loop použitý aj pri pridávaní blokov do front metódou `void async_after(Dispatch::Workitem* block, uint32_t ms)`.

Threadpool udržiava kolekciu udalostí, ktoré su vygenerované týmito metódami. Pri vytvorení vlákna je nastavený jeho spôsob plánovania na typ SCHED_RR a jeho priorita je vyššia ako vláken vykonávajúcich prácu User Interactive. Vyššia priorita je potrebná pre zabezpečenie, aby bol čas čo najpresnejší. Spomínaná condition variable podporuje aj metódu `wait_until`. Po ubehnutí zvoleného času je vlákno zobudené a vďaka nastavenej priorite vyvolá nedobrovoľnú zmenu kontextu pre pracujúce vlákno v objekte threadpool.

```
auto *timer = new SourceTimer(global);
timer->set_handler([start]{
   printf("timer block\n");
});
timer->schedule(0,1000);
timer->resume();
```

Výpis 4.3: Použitie Dispatch Source Timer

4.5 Skupiny a semafóry

Entity využívané na synchronizáciu blokov sú implementované využtím `std::atomic` ako počítadlo. Objekty typu atomic definujú dátové typy pri ktorých nevzniká neočakávané chovanie, keď jedno vlákno číta hodnotu a druhé zapisuje. Využitím atomického počítadla nie je potrebné používať zámok pre kritickú sekciu skupín, semafórov a interného počítadla iterácii v metóde `Dispatch::Queue.concurrent_perform`.

Implementácia ako knižnica

Systém je implementovaný ako knižnica pre jazyk C++. Pre preklad je využitý nástroj CMake[2]. CMake je open-source, multi-platformový nástroj pre preklad. Knižnica obsahuje jeden hlavičkový súbor, ktorý zapúzdruje celé rozhranie knižnice. Taktiež je zahrnutá zložka *examples*, ktorá obsahuje zdrojové súbory využité na vyhodnotenie fungovania knižnice. Ostatné príklad poukazujú na štýl programovania využitím systému Grand Central Dispatch a všetkých jeho možností.

Rozhranie knižnice je taktiež zdokumentované štýlom kompatibliným s nástrojom Doxygen[3]. Tento nástroj je štandardom pre generovanie dokumentácie zo zdrojového kódu jazyka C++. Je možné ju nájst v zložke *doc*.

[2] https://cmake.org
[3] http://www.doxygen.nl

Kapitola 5

Vyhodnotenie implementovanej knižnice

Po tom ako bol systém Grand Central Dispatch implementovaný ako knižnica v jazyku C++ bol vyhodnotený jej výkon a korektnosť správania priorít a paralelizmu.

Prvé vyhodnotenie sa sústreďuje na výkon na systémoch s roznym počtom procesorom vzhľadom na čas. Ďalej je vyhodnotená réžia knižnice pri spracovávaní blokov použitím profilovania. Posledné vyhodnotenie ukazuje fungovanie vlastnosti Quality of Service a jej dopad na výkon.

Ďalej sú v tejto kapitole priblížené príklady využitia knižnice a jeho entít.

5.1 Réžia spojená s vykonávaním blokov

Systém Grand Central Dispatch abstrahuje riadenie vláken od programátora a preto prináša aj istú réžiu pri spracovaní a vykonaní blokov. Réžia bola zmeraná využitím nástroja **dtrace** vo vývojovom prostredí CLion. Pomocou tohto nástroja je možné profilovať beh programu. So zvolenou hodnotou vzorky určuje v ktorej časti kódu je program najdlhšie.

Profilovanie nástrojom dtrace prebiehalo na zariadení MacBook Pro s procesorom 2,9 GHz Dual-Core Intel Core i5 a operačným systémom macOS Catalina 10.15.4.

Zvolené vzorkovanie bolo 5000 vzoriek za sekundu, čo znamená, že jedna vzorka je približne 0.2ms.

Keďže hlavným rozhraním systému je pridávanie blokov do fronty, bola zmeraná metóda asynchronného pridávania blokov do fronty. Keďže operácia netrvá rádovo v milisekundách, profilovanie prebehlo nad väčším počtom blokov.

Počet volaní metódy async	počet vzoriek	ubehnutý čas	jedno volanie async
1,000,000	7,110	1.42s	1.42ns
1,000,000	7,469	1.49s	1.49ns
1,000,000	6,872	1.37s	1.37ns
10,000,000	65,815	13.16s	1.31ns
10,000,000	71,938	14.39s	1.44ns
10,000,000	74,536	14,90s	1.49ns

Z tabuľky je možno vidieť, že po profilovaní metódy pri väčšom počte blokov trvá volanie metódy async nad frontou približne 2.5 až 3 nanosekundy.

Ďalším dôležitým bodom systému je réžia na úrovni jednotlivých vláken objektu threadpool. Tie implementujú logiku front a priorít. Nasledujúca tabuľka zobrazuje réžiu jednotlivého vlákna pri spracovaní bloku. Je dôležité podotknúť, že réžia nie je nijak spätá s obsahom vykonávaného bloku. Profilovanie taktiež prebehlo nad väčším počtom blokov.

	Počet blokov	počet vzoriek	celkový ubehnutý čas	réžia bloku
hlavná fronta	1,000,000	9,323	1.86s	1.86ns
hlavná fronta	10,000,000	84,971	17.00s	1.70ns
threadpool	1,000,000	32,680	6.54s	6.54ns
threadpool	10,000,000	311,681	62.34s	6.23ns

Ako tabuľka naznačuje, réžia spracovania bloku z fronty typu concurrent je náročnejšia ako réžia hlavnej fronty. Je to kvôli tomu, že vlákna threadpoolu implementujú zložitejšiu logiku front ako hlavné vlákno systému. Čas v stĺpci celkového ubehnutého času je rozdelený medzi všetky procesory. Čas spojený s réžiou jedného bloku na jednom procesore je teda približne 6.5 nanosekundy. Réžia na hlavnom vlákne je niekde medzi 1.5 a 2 nanosekundami.

Porovnanie návrhového vzoru threadpool a vytvárania vláken

Ako bolo spomínané v kapitole 2 najväčšou výhodou návrhového vzoru threadpool je jeho efektivita. Pri znovu-použití vláken odpadá réžia vytvorenia a odstránenia vlákna pri každom bloku. Nasledujúca tabuľka porovnáva predošlé hodnoty réžie zmerané pri pridaní bloku do fronty, réžiu jednotlivého vlákna pri spracovaní bloku a réžiu potrebnú na vytvorenie a odstránenie vlákna.

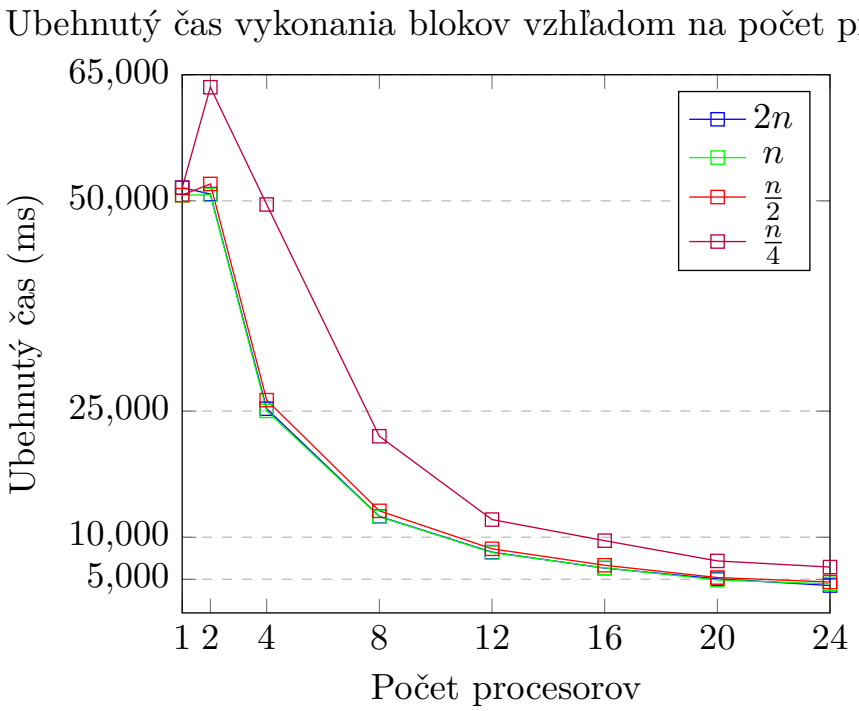
	Počet blokov/vláken	ubehnutý čas	jedno volanie
réžia GCD fronty	1,000,000	1.42s	1.42ns
réžia GCD vlákna	1,000,000	6.54s	6.54ns
réžia nových vláken	1,000,000	23.87s	23.87ns

Tabuľka ukazuje priemerné hodnoty po viacerých meraniach. Z meraní je jasne vidieť efektivitu návrhového vzoru threadpool. Réžia pridávania blokov do front a vytvárania nových vláken prebieha na jednom procesore, zatiaľ čo réžia konkrétnych GCD vláken je prerozdelená medzi procesory. Preto možeme považovať réžiu implementovaného systému približne 5-10x efektívnejšiu ako pri vytváraní nového vlákna pre každý blok.

5.2 Vyhodnotenie knižnice pri rôznom počte procesorov

V predošlích meraniach bolo využité jedno zariadenie so štyrmi logickými vláknami. V nasledujúcom vyhodnotení boli vytvorené virtuálne počítače s rôznym počtom procesorov (konkrétne typ n1-standard na platforme Google Cloud Platform[1]) s operačným systémom Debian GNU/Linux 9.

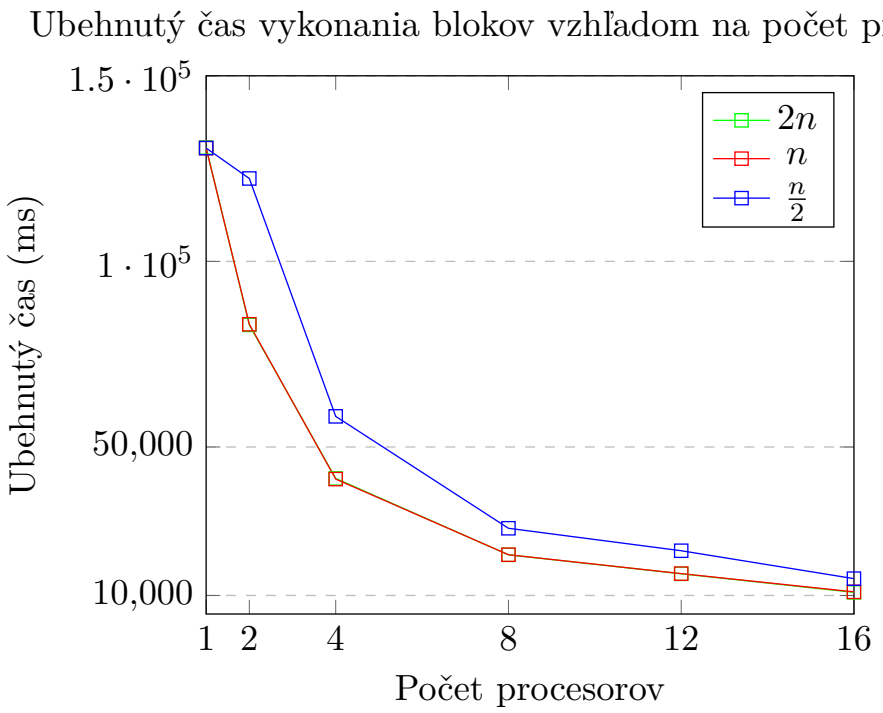
Pri testovaní sa do fronty typu concurrent pridali výpočetne náročné bloky a bol zmeraný čas potrebný na ich vykonanie. Testované boli 2 typy blokov. Prvý graf ukazuje výpočetne náročnú operáciu násobenia matíc o veľkosti 500x500, ktorá je ale náročna aj pre pamäť RAM a cache. Drúhy graf, ukazuje náročný výpočet, ktorý je menej závislý na pamäti.



Jednotlivé funkcie na grafe zobrazujú počet pracujúcich vláken objektu threadpool, kde N = počet procesorov. Tabuľka s konkrétnymi hodnotami je v prílohe A.

Z grafu je možné vidieť, že N a $\frac{1}{2}N$ majú podobnú, takmer rovnakú dobu na vykonanie úloh. Násobenie matice je závislé aj na pamäti RAM. Preto je čas vykonania pri dvoch a jednom procesore takmer rovnaký, tak isto ako pri N a $\frac{1}{2}N$ vláknach. Na ďalšom grafe je ukázaný iný typ bloku, ktorý nie je závislý na pamäti RAM, cache ani I/O operácii.

[1] https://cloud.google.com



Na druhom grafe je možné vidieť, že operácie závislé hlavne na procesore sú efektívnejšie, ak je vlákno vytvorené pre každý procesor. Z prvého a druhého grafu vyplýva, že pridaním vláken nad počet procesorov nezvyšuje efektivitu vykonávaného kódu. Správne použitie Quality of Service pri využití knižnice môže zefektívniť prácu, ktorá potrebuje aj iné zdroje ako procesor. Rovnako ako pri prvom meraní je možné nájst konkrétne hodnoty v prílohe A.

5.3 Vyhodnotenie dopadu Quality of Service na vykonanie blokov

Hodnota Quality of Service bola implementovaná zmenou konfigurácie plánovača pomocou rozhrania POSIX vláken. Kód na výpise 5.1 demonštruje vynútenú zmenu kontextu po pridaní úlohy s vyššou prioritou. Ako bolo spomenuté vyššie v objekte threadpool je $2N$ vláken, kde N je počet procesorov a polovica vytvorených vláken je vyhradená pre úlohy s vyššou prioritou ako práve vykonávané bloky. Demonštračný kód bol spustený na zariadení s jedným procesorom, ale vynútená zmena kontextu funguje rovnako, keď sú na viac-jadrových zariadeniach zamestnané všetky jadrá. Pod kódom je výpis zo štandardného výstupu. Môžeme vidieť, že hoci blok s prioritou Default bol do fronty pridaný neskôr, tak jeho vykonanie skončilo skôr ako už vykonávaný blok s prioritou Background.

```
using namespace Dispatch;
auto it = get_global_queue(DISPATCH_QOS_USER_INITIATED);
auto ui = get_global_queue(DISPATCH_QOS_USER_INTERACTIVE);
unsigned long total_start = SourceTimer::now();
bg->async([&,total_start]{
   printf("BACKGROUND START %lums\n", SourceTimer::now() - total_start);
   process();
   printf("BACKGROUND END %lums\n", SourceTimer::now()-total_start);
});
df->async_after([&,total_start]{
   printf("DEFAULT START %lums\n", SourceTimer::now() - total_start);
   process();
   printf("DEFAULT END %lums\n", SourceTimer::now()-total_start);
}, 500);


//BACKGROUND START 0ms
//DEFAULT START 501ms
//DEFAULT END 3118ms
//BACKGROUND END 5223ms
```

Výpis 5.1: Dopad Quality of Service na vykonanie blokov

Keďže systém Grand Central Dispatch sa používa najmä pri systémoch a aplikáciach, ktoré potrebujú rýchlu odozvu, je správne využívanie hodnoty Quality of Service dôležité.

Systémy Linux podporujú iba hodnotu PTHREAD_SCOPE_SYSTEM. To znamená, že plánujú všetky vlákna spoločne, nezávisle na procese ku ktorému patria. Preto bloky s prioritou User Initiated a User Interactive môžu mať vplyv na ostatné procesy bežiace na zariadení.

5.4 Príklady použitia knižnice

V tejto sekcii sú popisané príklady ako sa knižnica Grand Central Dispatch používa. Príklady sú napísané v jazyku C++ a reprezentujú použitie implementovanej knižnice.

Systém Grand Central Dispatch je hlavne využívaný pri systémoch riadených udalosťami a jeho návrhové vzory to aj odzrkadlujú. Nasledujúce príklady ukazujú všeobecné využitie základných entít knižnice.

Bariérový blok

Bariérový blok sa používa pri prístupe ku kritickej sekcii. Pri vytvorení súkromnej fronty typu concurrent pre prístup k dátam je možné využiť bariéru pre zápis a normálny typ bloku pre čítanie. To znamená, že ku kritickej sekcii môže naraz pristúpiť len jedno vlákno k zápisu, alebo viacero vláken naraz k čítaniu.

```
auto data_queue = new Dispatch::Queue(true,DISPATCH_QOS_DEFAULT);
data_queue->async([&]{
   //data write
},{true}); //true stands for barrier
data_queue->async([&]{
   //data read
}); //multiple blocks can access data to read
```

Výpis 5.2: Použitie bariéry

Paralelný cyklus

Metóda `concurrent_perform` slúži na vykonanie cyklu na viacerých vláknach. Pri cykloch, ktorých iterácie môžu bežať paralelne tento spôsob zvyšuje efektivitu.

```
auto q = new Dispatch::Queue(true,DISPATCH_QOS_DEFAULT);
//runs this loop in paralel(concurrent) instead of running on single thread
q->concurrent_perform(size,[&](int i){
   for (int j = 0; j < size; j++) {
      mult(a, b, size, i, j, result);
   }
});
```

Výpis 5.3: Paralelný cyklus

Pri veľkom počte iterácii, kde každá vykonáva výpočetne nenáročnú úlohu je metóda `concurrent_perform` efektívnejšia pretože eliminuje réžiu pridávania blokov do fronty pri využití metódy `async` v cykle.

Dispatch Semaphore

Semafórom je možné obmedziť prístup ku kritickej sekcii v programe. Môže nastať situácia, kedy je potrebné pristúpiť ku kritickej sekcii z viacerých blokov. Keďže programátor nevie s istotou povedať, kedy sú bloky vo fronte typu concurrent vykonávané, musí využiť semafór.

```
auto q = Dispatch::get_global_queue(DISPATCH_QOS_BACKGROUND);
Dispatch::Semaphore semaphore(1);
//note: tasks can be enqueued from different parts of code
q->async([&]{
   semaphore.wait();
   //access to critical section
   semaphore.signal();
});
q->async([&]{
   semaphore.wait();
   //access to critical section
   semaphore.signal();
});
```

Výpis 5.4: Použitie Dispatch Semaphore

Dispatch Group

V prípade, že je možné viacero blokov považovať za istú skupinu, využije sa Dispatch Group. Nasledujúci príklad ukazuje ako do hlavnej fronty pridať blok po tom, ako sa všetky bloky v skupine vykonali.

```
auto q = Dispatch::get_global_queue(DISPATCH_QOS_BACKGROUND);
auto main = Dispatch::get_main_queue();
Dispatch::Group group;
//note: tasks can be enqueued from different parts of code
group.enter() //tasks enter the group before enqueuing
q->async([&]{
   //download file1
   group.leave();
});
group.enter()
q->async([&]{
   //download file2
   group.leave();
});
//enqueue another task after blocks finish
group.notify(main,[&]{printf("all files downloaded");});
//thread is blocked until all tasks finish
group.wait()
```

Výpis 5.5: Použitie Dispatch Group

Serial Queue deadlock

V nasledujúcom príklade je ukázaný zlý spôsob použitia fronty typu serial vrátane hlavnej Dispatch Queue. Vzniká deadlock, kedy vlákno vykonávajúce prvý blok vlastní a zároveň čaká na uvoľnenie zámku.

```
//main queue is serial queue
auto main = Dispatch::get_main_queue();
main->async([main]{//enqueue task1 to main queue
   main->sync([]{printf("deadlocked");});//task2 enqueued to serial queue
   //this task starts executing when task1 finishes
   //task1 finishes when task2 finishes
});
```

Výpis 5.6: Deadlock v hlavnej fronte

Server s použitím funkcie select

Implementovaný systém Grand Central Dispatch je navrhnutý pre efektívne využitie procesorov. Pri pridaní bloku, ktorý čaká na I/O operáciu je vlákno v objekte threadpool zastavené. Pri pridaní viacerých blokov čakajúcich na I/O, mutex alebo časový interval je možné vlákna úplne odstaviť. Preto je potrebné nájsť spôsob monitorovania, ktorý určí kedy sa blok s úlohou, ktorá využíva procesor do fronty zaradí.

Implementovaný Dispatch Source Timer slúži na pridávanie blokov do front po zvolenom čase bez toho, aby boli pracujúce vlákna blokované počas tejto doby. Nasledujúci príklad ukazuje možné využitie systému pri využití funkcie `select`[2]. Táto funkcia monitoruje zmenu na deskriptore súboru (napríklad soket). Pri zmene je možné prečítať dáta a spracovať ich v bloku odovzdanému do fronty.

Nasledujúci výpis popisuje pseudo-kód možného využitia systému pre spracovanie viacerých klientov naraz, resp. potencionálny prototyp pre ďalší typ objektu Dispatch Source.

```
auto q = Dispatch::get_global_queue(DISPATCH_QOS_DEFAULT);
while(true){
   select( max_sd + 1 , &readfds , NULL , NULL , NULL);
   for(int i = 0; i < clients_count; i++){
      if(FD_SET(clients[i],&readfs)){
         if(read(sd,buffer,LENGTH) == 0){
            //handle error
         }else{
            q->async([&]{
               //handle client request
            });
         }
      }
   }
}
```

Výpis 5.7: Využitie GCD s funkciou select ako server

[2] http://man7.org/linux/man-pages/man2/select.2.html

Kapitola 6

Záver

Systém Grand Central Dispatch zjednodušuje programátorom správu súbežného kódu tým, že poskytuje rozhranie front a blokov, ktoré sú vykonávané interne alokovaným objektom threadpool. Taktiež poskytuje určovanie priority odovzdaných blokov a synchronizačné prostriedky pre ich vykonanie. V prvej časti práce je priblížený existujúci systém a základ viacvláknového programovania.

V ďalšej časti je priblížený hlavný cieľ práce. Cieľom práce bolo implementovať knižnicu v jazyku C++, ktorá čo najlepšie replikuje funkcionalitu tohto systému. Za využitia štandardu jazyka C++ a rozhrania POSIX vláken je knižnica prenositeľná medzi operačnými systémami podporujúcimi pthreads.

Po návrhu a implementácii bola knižnica vyhodnotená profilovaním a testovaním správnosti synchronizačných prvkov. Taktiež bola vyhodnotená v prostrediach s rôznym počtom procesorov. Réžia nameraná profilovaním implementácia bola pomerne nízka. Knižnica je zdokumentovaná nástrojom Doxygen a zabalená nástrojom CMake. Oba tieto nástroje sú štandardom pri pracovaní s jazykom C++.

Knižnicu je možné ďalej vyvíjat smerom k zameraniu na aplikácie riadené udalosťami. To pozostáva z pridania rôznych typov objektu Dispatch Source alebo iných možností pridávania blokov do front pri systémových udalostiach. Taktiež je možné upraviť objekt threadpool pre dynamické alokovanie nových vláken pri situáciách, kedy alokované vlákna čakajú na I/O operáciu alebo iný zdroj. Ďalej by threadpool mohol rozhodovať o počte alokovaní vláken na základe stavu zariadenia a hodnôt Quality of Service.

Literatúra

[1] *C++ reference* [online]. [cit. 2020-27-05]. Dostupné z: https://en.cppreference.com/w/cpp.

[2] *Grand Central Dispatch documentation* [online]. Apple Inc. [cit. 2020-11-05]. Dostupné z: https://developer.apple.com/documentation/DISPATCH.

[3] *Modernizing Grand Central Dispatch Usage* [online]. 2017 [cit. 2020-27-05]. Dostupné z: https://developer.apple.com/videos/play/wwdc2017/706/.

[4] Guntheroth, K. *Optimized C++: Proven Techniques for Heightened Performance.* 1st. O'Reilly Media, Inc., 2016. ISBN 1491922060.

[5] Kerrisk, M. *Linux man pages online* [online]. [cit. 2020-27-05]. Dostupné z: https://www.man7.org/linux/man-pages/index.html.

[6] Nichols, B., Buttlar, D. a Farrell, J. P. *Pthreads Programming.* USA: O'Reilly & Associates, Inc., 1996. ISBN 1565921151.

Príloha A

Hodnoty merania pri rôznom počte procesorov

V nasledujúcich tabuľkách sú ukázané presné namerané dáta z testov na viacerých procesoroch. Dáta sú v milisekundách. Riadky ukazújú počet pracujúcich vláken vzhľadom na počet procesorov.

V prvej tabuľke sú namerané hodnoty testu *dispatch/examples/matrix_mult/main.cpp*.

	1	2	4	8	12	16	20	24
N	50680	50723	25018	12443	8179	6361	4916	4462
$2N$	51568	50811	25281	12482	8228	6329	5059	4259
$\frac{N}{2}$	50680	52016	26306	13138	8623	6668	5208	4675
$\frac{3}{4}N$	50680	52016	26023	12825	8433	6519	5272	4469

V ďalšej tabuľke sú namerané hodnoty druhého testovacie programu z *dispatch/examples/cpu_bound/main.cpp*. Tento test v iterácii používal aritmetické a goniometrické operácie.

	1	2	4	8	12	16	20	24
N	130524	83065	41355	20976	15883	10967	8268	7762
$2N$	130676	82888	41528	20990	15835	10833	8710	7215
$\frac{N}{2}$	130524	112336	56251	28100	22049	14530	11227	10143
$\frac{3}{4}N$	130524	112336	48489	24100	18824	12463	9856	8347