UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Igor Bišćanin

VPLIV NITNIH BAZENOV NA ZMOGLJIVOST APLIKACIJ

DIPLOMSKO DELO
VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00083/2011
Datum: 01.04.2011

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **IGOR BIŠĆANIN**

Naslov: **VPLIV NITNIH BAZENOV NA ZMOGLJIVOST APLIKACIJ**

THE INFLUENCE OF THREAD POOLS ON APPLICATION PERFORMANCE

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija prve stopnje

Tematika naloge:

Večnitne aplikacije veljajo za hitrejše, saj lahko učinkovito izkoriščajo zmožnosti sodobnih računalniških sistemov. V diplomski nalogi predstavite principe večnitnosti in na praktičnem primeru prepisovanja večjega števila datotek raziščite učinkovitost in meje uporabe različnih nitnih bazenov. V ta namen razvijte ustrezno programsko opremo in izvedite čimbolj celovito primerjavo zmogljivosti nitnih bazenov. Nalogo zaključite z analizo rezultatov.

Mentor:

viš. pred. dr. Igor Rožanc

Dekan:

prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko
2
UNIVERZA V LJUBLJANI

Ljubljana, 2011

I Z J A V A O A V T O R S T V U

diplomskega dela

Spodaj podpisani/-a Igor Bišćanin,

z vpisno številko 63010011,

sem avtor diplomskega dela z naslovom:

Vpliv nitnih bazenov na zmogljivost aplikacij

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal/-a samostojno pod mentorstvom

  viš. pred. dr. Igorja Rožanca,

- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,

- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki »Dela FRI«.

V Ljubljani, 18. 10. 2011 Podpis avtorja:

Zahvala

Ob zaključku študija, ki je trajal dlje, kot je bilo pričakovano, se iz srca zahvaljujem svojemu mentorju, višjemu predavatelju Igorju Rožancu za vso pomoč, ki mi jo je nudil ob nastajanju diplomskega dela, ter za prijetno sodelovanje in precejšnjo mero potrpežljivosti in razumevanja.

Zahvaljujem bi se tudi sodelavcem v podjetju TopIT d. o. o. za vso strokovno pomoč in znanje, ki so mi ga dali.

Posebej pa se zahvaljujem svoji družini in dekletu, ki so me v dolgih letih študija podpirali in spodbujali ter mi v težkih trenutkih stali ob strani.

Kazalo

Slike

Tabele

Povzetek

Večnitne aplikacije veljajo za hitrejše, saj lahko bolj učinkovito izkoriščajo zmožnosti sodobnih računalniških sistemov.

Cilj diplomskega dela je z enostavno Windows aplikacijo za prepisovanje datotek iz enega v drug imenik z uporabo različnih nitnih bazenov predstaviti učinkovitost večnitnih aplikacij.

V diplomskem delu smo predstavili načine razvoja večnitnih aplikacij, nekatere omejitve, na katere je potrebno biti pri razvoju večnitnih aplikacij pozoren, in principe zaščite dostopa do deljenih podatkov sočasno iz različnih niti.

V okolju Visual Studio 2010 smo s programskim jezikom C++ razvili konzolno aplikacijo za prepisovanje datotek. Aplikacija na podlagi vhodnih podatkov določi število delovnih niti v nitnem bazenu, ki jim potem dodeljuje posamezne naloge.

Učinkovitost aplikacije smo testirali z različnimi testnimi primeri in rezultate tudi ustrezno predstavili.

Ugotovili smo, da so večnitne aplikacije učinkovitejše pri prepisovanju večjih datotek, čeprav trdi disk predstavlja določeno omejitev.

Pri prepisovanju manjših datotek, ki je hitrejša operacija, smo ugotovili, da večje število delovnih niti v nitnem bazenu ne poveča učinkovitosti aplikacije. Čas, potreben za kreiranje novih delovnih niti, in čas, potreben za preverjanje, ali je kakšna delovna nit prosta, namreč zmanjšata učinkovitost aplikacij, če je delovnih niti preveč.

Ključne besede: nitni bazen, primerjava zmogljivosti, večnitnost, sočasnost, sinhronizacijski objekti

Abstract

Multithreaded applications are faster as they can effectively exploit the capabilities of modern computer systems.

The purpose of this thesis is to present the efficiency of multithreaded applications with the use of a simple Windows console application which copies files from one directory to another using different thread pools.

In this thesis we describe the ways of developing multithreaded applications, some limitations which should be taken into account when developing multithreaded applications, and the principles of protecting the access to shared data from different threads simultaneously.

A console application for the copying of files was developed in Visual Studio 2010 in C++ programming language. Based on the input parameter, the application determines the number of working threads in a thread pool and then assigns a specific task to each thread.

The efficiency of the application was tested with various test cases and the results are suitably presented.

We learned that multithreaded applications are more efficient for the copying of larger files although the hard disk presents a certain limitation.

A higher number of worker threads does not increase the efficiency of the application when copying smaller files which is a faster operation. The time needed for creating new worker threads and for checking whether any threads are free decreases the efficiency of the application if the number of worker threads is too high.

Key Words: thread pool, performance comparison, multithreading, concurrency, synchronization objects

1 Uvod

Veliko novejših aplikacij za večjo učinkovitost in hitrejše izvajanje na računalniškem sistemu uporablja več niti ali pa celo več procesov. Take aplikacije imenujemo večnitne aplikacije (ang. multithreading applications). Večina večnitnih aplikacij uporablja za upravljanje z nitmi zelo razširjene in uporabljene tako imenovane nitne bazene (ang. thread pools).

Nitni bazen ima v primerjavi z ustvarjanjem nove niti za vsako nalogo ali operacijo dve večji prednosti. Prva prednost je večja učinkovitost delovanja aplikacije ali sistema zaradi odprave porabe časa in drugih sistemskih virov pri kreiranju in uničevanju niti. Druga večja prednost pa je boljša uporaba sistemskih virov z dinamičnim kreiranjem števila niti glede na število nalog ali operacij, ki jih mora aplikacija izvesti. V primeru večjega števila nalog tako aplikacija ustvari dodatne niti, ki jih potem, ko opravijo svojo nalogo, tudi uniči. Take niti imenujemo nestalne delovne niti. V nitnem bazenu se tako upravlja s stalnimi (ang. persistent threads) in nestalnimi (ang. nonpersistent threads) delovnimi nitmi.

Velik problem pri razvoju večnitnih aplikacij oziroma nitnih bazenov predstavlja določitev števila posameznih tipov niti v nitnem bazenu, ki so potrebne za najboljše oziroma najhitrejše delovanje aplikacije oziroma njenih operacij. Problem natančneje predstavlja izračun števila tako imenovanih stalnih oziroma nestalnih delovnih niti v nitnem bazenu in izračun s tem povezanega največjega števila vseh delovnih niti v nitnem bazenu.

Mnoge aplikacije med svojim delovanjem izvajajo različne tipe nalog ali operacij, ki glede na prioriteto za svoje optimalno izvajanje lahko potrebujejo različno število stalnih in tudi nestalnih delovnih niti.

Namen diplomskega dela je z enostavno aplikacijo za kopiranje datotek predstaviti delovanje nitnega bazena in primerjava rezultatov kopiranja datotek z različnim številom stalnih in nestalnih delovnih niti v nitnem bazenu. S primerjavo rezultatov želimo določiti način uporabe optimalnega števila delovnih niti za delovanje naše aplikacije.

Aplikacija je razvita z objektnim programskim jezikom C++ v okolju Visual Studio 2010. Za delo z nitmi se uporablja razširjene razrede abstrakcije `OMNIThread`.

Pri določitvi optimalnih vrednosti števila niti pa je treba biti pozoren tudi na določene omejitve. Prvo vrsto omejitev predstavljajo sistemske omejitve, med katere uvrščamo CPE (centralno procesno enoto, ang. central processing unit), RAM (ang. Random Access memory; slo. Bralno pisalni pomnilnik), največje možno število niti v enem procesu, branje z diska, pisanje na disk. Drugo vrsto omejitev pa predstavlja aplikacija sama oziroma njen delež pri obremenitvi celotnega sistema (skupno število niti aplikacije na sistemu, poraba delovnega pomnilnika in centralne procesne enote).

2 Sočasnost in večnitnost

Za večjo učinkovitost izvajajo novejše aplikacije na računalniškem sistemu več nalog sočasno, pri tem pa uporabljajo več niti ali pa celo več procesov.

Ker lahko pri sočasnem izvajanju nalog več niti dostopa do istih podatkov, je treba podatke v času obdelave zaščititi, za kar uporabljamo sinhronizacijske objekte (ang. synhronization objects).

V primerih, ko več niti čaka ena drugo, da sprostijo dostop do zaščitenih podatkov, nastane situacija, ko program ne more nadaljevati izvajanja nalog. Tako situacijo imenujemo smrtni objem (ang. deadlock).

Za ustvarjanje in upravljanje niti ter dodeljevanje nalog delovnim nitim aplikacije največkrat uporabljajo nitne bazene (ang. thread pools).[5]

2.1 Sočasnost

Sočasnost ali sočasno izvajanje je stanje, ko se dve ali več stvari dogaja oz. izvaja ob istem času na istem mestu.[1]

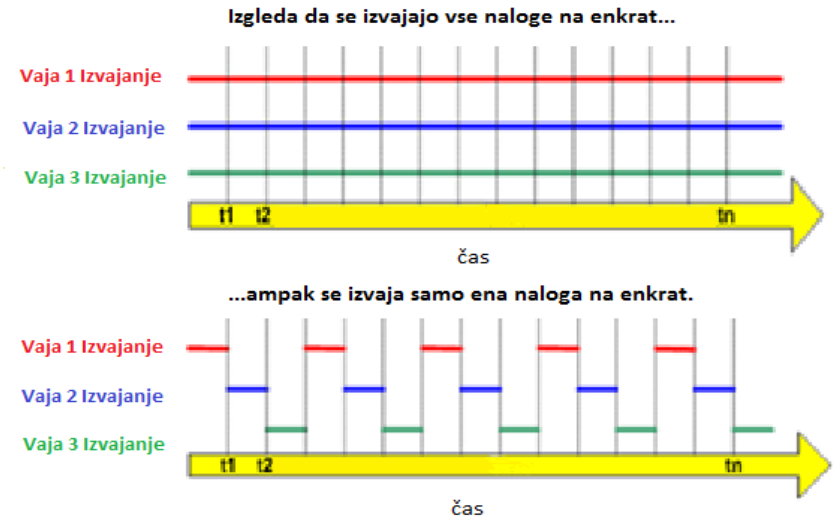
Ko govorimo o pojmu sočasnost v svetu računalništva in programske opreme, govorimo o sistemu ali aplikaciji, ki hkrati oziroma sočasno (v istem trenutku) izvaja neodvisne aktivnosti, naloge oziroma operacije.[2]

Sočasnost (ang. concurrency) je pojem, ki se v računalniški terminologiji tudi najpogosteje uporablja za opis hkratnega izvajanja več nalog.[2]

Operacijski sistemi so že mnogo let znani po tem, da lahko izvajajo naloge različnih aplikacij sočasno, tako da preklapljajo med nalogami in vsaki dodelijo določen čas za izvajanje.

Računalniki z enim procesorjem v resnici izvajajo le eno nalogo hkrati, vendar preklapljajo med izvajanjem različnih nalog večkrat v sekundi, tako da teh preklapljanj uporabnik skoraj ne opazi. Preklapljanje med izvajanjem različnih nalog imenujemo z angleškim izrazom task switching. Kljub temu še vedno govorimo o sočasnosti, saj je preklapljanje med nalogami tako hitro, da ne vemo, kdaj se naloga izvaja in kdaj naloga čaka, da se bo spet začela izvajati.[2]

Na sliki 1 je prikazan način preklapljanja med nalogami.

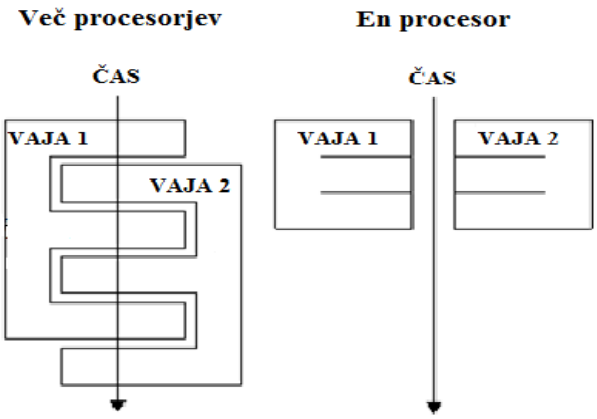


Slika 1: Preklapljanje med izvajanjem nalog [9]

Z razvojem računalniških sistemov z več procesorji in z razvojem procesorjev z več jedri se je hitrost in število izvajanj teh nalog še povečala. Računalniki z več kot enim procesorjem in računalniki z več jedri v enem procesorju namreč lahko izvajajo več nalog hkrati. Izvajanje več nalog hkrati s pomočjo strojne opreme imenujemo sočasnost strojne opreme (ang hardware concurrency).[1]

Z razvojem strojne opreme pa se je spremenil tudi način razvoja programske opreme. Preden se je v računalniških sistemih začela uporaba več procesorjev ali pa procesorjev z več jedri, je lahko aplikacija izvajala samo eno nalogo naenkrat in samo ta naloga je lahko dostopala do skupnih oziroma deljenih podatkov. Z uporabo več procesorjev oziroma procesorjev z več jedri pa lahko aplikacije izvajajo več nalog hkrati in tako lahko več nalog iz različnih niti dostopa do skupnih podatkov. Če želimo ali moramo ohraniti konsistentnost skupnih podatkov, je treba preprečiti dostopanje do istih podatkov istočasno iz različnih niti. To pa lahko storimo samo z razvojem večnitnih aplikacij oziroma programske opreme.

Na sliki 2 vidimo razliko med izvajanjem nalog na sistemu z enim procesorjem in na sistemu z več procesorji.



Slika 2: Izvajane naloge na sistemu z enim in na sistemu z dvema procesorjema [2]

K razvoju večnitnih aplikacij lahko pristopimo na dva načina. En način razvoja aplikacij oziroma sistemov, ki izvajajo naloge sočasno, je, da aplikacija za opravljanje svojih nalog uporablja več enonitnih procesov. Drugi način razvoja pa je tak, da aplikacija za opravljanje več svojih nalog sočasno uporablja več niti, ki so v enem samem procesu. Oba omenjena načina razvoja pa lahko tudi združimo in uporabimo več procesov, ki so lahko enonitni ali pa večnitni. Vsak izmed omenjenih načinov razvoja večnitnih aplikacij pa ima svoje prednosti in slabosti.[2]

Ena izmed slabosti pristopa razvoja večnitnih aplikacij z več enonitnimi procesi je komunikacija med procesi. Komunikacijo je zapleteno vzpostaviti, ko pa jo vzpostavimo, je lahko počasna. Operacijski sistemi sami v medprocesni komunikaciji uporabljajo veliko zaščite, saj želijo s tem preprečili enemu procesu spremembo podatkov drugega procesa, kar predstavlja dodatne težave pri izvedbi.

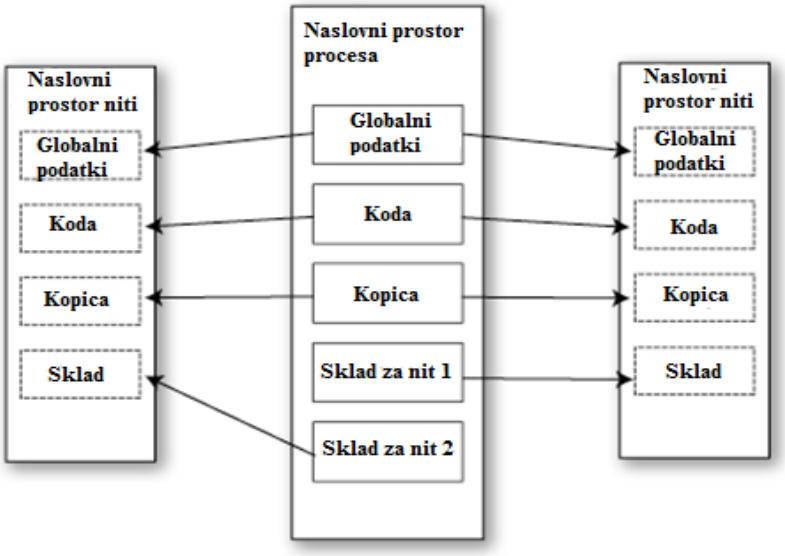
Drugo slabost pristopa razvoja večnitnih aplikacij z več enonitnimi procesi predstavljajo sistemski viri in čas, ki je potreben za kreiranje in zagon vsakega posameznega procesa. Vsak nov enonitni proces porabi več sistemskih virov kot vsaka posamezna nova nit v enem procesu.

Nima pa pristop razvoja večnitnih aplikacij z več enonitnimi procesi samih slabosti. Zgoraj omenjena zaščita, ki jo operacijski sistemi uporabljajo v medprocesni komunikaciji, omogoča razvoj varnejše kode, saj je skupne oziroma deljene podatke iz drugega procesa težje spreminjati. Uporaba več procesov na različnih sistemih, ki so povezani preko omrežja, pa predstavlja predvsem prednost v učinkovitosti delovanja.

Pri pristopu razvoja večnitnih aplikacij z več nitmi v enem procesu, je nit enaka enostavnemu procesu, ki opravlja svoje naloge neodvisno od drugih procesov. Za razliko od procesov si vse niti v procesu delijo skupni naslovni prostor. Večina podatkov v naslovnem prostoru je dostopna direktno iz niti, zato ni treba vzpostavljati komunikacije med nitmi, kot jo je treba vzpostavljati med procesi.

V primerjavi z več enonitnimi procesi deljen naslovni prostor in manjša zaščita podatkov v večnitnih procesih zmanjša del virov, ki se porabijo za zagotavljanje varnosti, ki je potrebna za komunikacijo med procesi.

Na sliki 3 je prikazan deljen naslovni prostor (ang. shared address space) procesa.



Slika 3: Deljen naslovni prostor [11]

Vendar pa imata deljen naslovni prostor procesa in manjša zaščita podatkov v komunikaciji med nitmi tudi svojo ceno. Razvijalec mora biti pri razvoju zelo pozoren in natančen, saj morajo biti skupni oziroma deljeni podatki, do katerih lahko dostopajo različne niti sočasno, v času obdelave zaščiteni. Zaščita teh podatkov v času obdelave zahteva več časa pri razvoju in seveda testiranju, napisana koda pa ima lahko več potencialnih nepravilnosti oziroma napak pri izvajanju.

Kot že omenjeno je zaradi manj režije in krajšega časa razvoja pri razvoju večnitnih aplikacij pristop z večnitnim procesom bolj uporaben in velja za boljši način razvoja sočasnosti v programskem jeziku C++. Programski jezik C++ kot tak še ne zagotavlja podpore za med-procesno komunikacijo.[2]

Sočasnost se pri razvoju programske opreme večinoma uporablja, ko želimo ločiti izvajanje neodvisnih delov aplikacije ali pa kadar želimo doseči večjo zmogljivost oziroma hitrejše izvajanje večjega števila nalog iste aplikacije.

Z uporabo sočasnosti lahko hitrost delovanja aplikacije povečamo na dva načina.

Prvi način uporabe sočasnosti paralelizem nalog (ang. task parallelism) je način, da eno nalogo (operacijo) razdelimo na več manjših neodvisnih nalog, ki jih potem lahko izvajamo vzporedno. Tukaj predstavlja večino zahtevnosti pri razvoju ločevanje naloge na manjše, med

seboj neodvisne naloge. Ena nit lahko na primer opravi en del algoritma, druga nit pa sočasno izvede drugi del algoritma.

Drugi način uporabe sočasnosti pa uporablja pristop, s katerim na primer namesto nad eno datoteko izvajamo isto operacijo nad več različnimi datotekami sočasno. Prednost tega načina je v tem, da se porabi enako časa za izvajanje operacije nad eno ali pa nad več datotekami. Tak način angleško imenujemo paralelizem podatkov (ang. data parallelism).[2]

Pri razvoju večnitnih aplikacij in sistemov je zelo pomembno, da se zavedamo, kdaj sočasnost ne prinese bistvenega izboljšanja ali celo poslabša učinkovitost delovanja same aplikacije oziroma sistema.

Treba je premisliti in izračunati, ali bomo z dodatnim vložkom v razvoj sočasnosti res dosegli bistveno boljše rezultate. Koda, ki je napisana tako, da uporablja sočasnost, je velikokrat težja za razvoj, razumevanje in vzdrževanje, kar nam prinaša še dodatne stroške pri vzdrževanju same aplikacije oziroma sistema. Kompleksnost kode pa nemalokrat pomeni tudi več napak oziroma hroščev v delovanju naše programske opreme.

Druga stvar, ki je zelo pomembna, so sistemski viri, saj so niti omejeni viri. Če aplikacija med svojim delovanjem uporablja preveč niti, s tem porabi veliko sistemskih virov in celoten sistem postane zaradi tega počasnejši in manj odziven.

Preveč niti v enem procesu lahko porabi ves delovni spomin in skupni naslovni prostor, saj vsaka nit zahteva svoj prostor za sklad. Ta problem je sicer večji na 32-bitnih sistemih, kjer je omejitev spomina samo 4GB. Če ima torej vsaka nit 1MB prostora za sklad, kar je običajno za večino sistemov, je na 32-bitnem sistemu lahko največ 4096 niti. To pa v praksi ni mogoče doseči, saj se delovni spomin uporablja tudi za druge stvari. 64-bitni sistemi nimajo omejitve skupnega naslovnega prostora, vendar imajo druge omejitve, tako da bi preveliko število niti vsekakor povzročalo težave.

Več niti kot teče na sistemu, več vsebinskih preskokov mora sistem opraviti. Vsak tak preskok vzame nekaj časa, ki bi ga lahko sistem porabil za pomembnejše stvari. V določeni točki začne dodajanje novih niti in upravljanje velikega števila niti zmanjševati zmogljivost in učinkovitost aplikacije.

Uporaba sočasnosti v aplikacijah je torej smiselna le v kritičnih delih aplikacije, kjer lahko pridobimo na hitrosti in učinkovitosti izvajanja nalog aplikacije.

Določitev optimalnega števila stalnih in nestalnih niti je ena od nalog tega diplomskega dela. To bomo poskušali doseči s testiranjem in spreminjanjem vhodnih parametrov nitnega bazena, ki bodo določali število stalnih in nestalnih delovnih niti v nitnem bazenu.

2.2 Večnitnost

Večnitnost je posebna oblika večopravilnosti (ang. multitasking). Večopravilnost je sposobnost vzporedno izvajati opravila. V računalništvu večopravilnost omogoča operacijski

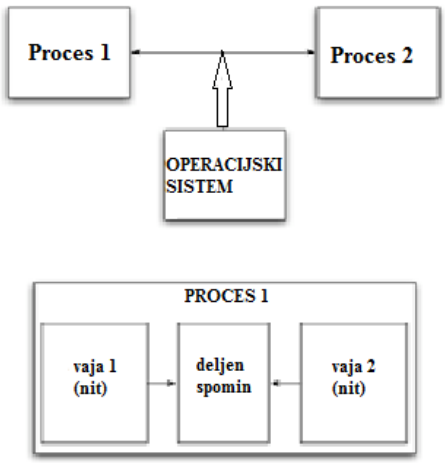
sistem. Poznamo dve vrsti večopravilnosti, procesno usmerjeno in nitno usmerjeno večopravilnost.

Procesno usmerjena večopravilnost je funkcionalnost, ki omogoča izvajanje več aplikacij ali več procesov sočasno, nitno usmerjena večopravilnost pa je funkcionalnost, ki omogoča sočasno izvajanje več operacij v enem procesu ene same aplikacije.

Večnitnost spremeni osnovno arhitekturo aplikacije, saj se večnitne aplikacije v primerjavi z enonitnimi aplikacijami ne izvajajo samo strogo zaporedno, ampak se določene operacije lahko izvajajo istočasno, kar pomeni, da lahko več operacij dostopa do istih podatkov v istem času.

Vsak proces vsebuje najmanj eno nit, ki se ustvari, ko se aplikacija zažene. Pravimo ji glavna nit in ustvari vse ostale niti v procesu.

Na sliki 4 je prikazana razlika med večopravilnostjo in večnitnostjo.



Slika 4: Večopravilnost in večnitnost [12]

Prednost večnitnosti je v tem, da omogoča uporabo prostega časa in prostih sistemskih virov, ki so prisotni pri izvajanju večine aplikacij. Večina vhodno izhodnih naprav (mrežnih vrat, diskov, tipkovnic) je počasnejših od centralno procesne enote, zato porabi program veliko časa za branje ali pošiljanje podatkov iz/do teh naprav.

Z uporabo večnitnosti lahko program v tem času opravi kakšno drugo neodvisno operacijo in s tem skrajša skupni čas izvajanja vseh operacij ali nalog aplikacije. En del aplikacije na primer pošilja datoteko preko interneta, drugi del programa pa lahko v tem času bere vhodne podatke iz tipkovnice ali bere podatke iz druge datoteke.

V primerjavi z Javo ali C# programski jezik C++, v katerem bo napisana naša aplikacija, nima vgrajene podpore za večnitnost in se v celoti zanaša na operacijski sistem in njegovo funkcionalnost za delo z nitmi. Ker je bil programski jezik C++ razvit za različne tipe

programiranja, bi vgrajena podpora za večnitnost, ki se od sistema do sistema razlikuje, pomenila preveliko omejitev. S tem bi jezik postal uporaben samo na takih sistemih, ki podpirajo večnitnost. S takim pristopom je C++ razvijalcem omogočil uporabo sistemskih funkcij, ki se od sistema do sistema razlikujejo. Operacijski sistem Windows ponuja bogat nabor funkcij za ustvarjanje in upravljanje niti.

Naloga diplomskega dela je razviti enostavno aplikacijo z enostavnim nitnim bazenom. V implementaciji bomo uporabljali svoje razrede, ki bodo razširjeni iz razredov abstrakcije `OMNIThread`.

`OMNIThread` abstrakcija je vmesnik, ki nam omogoča izvajanje različnih operacij za upravljanje z nitmi v programskem jeziku C++. Aplikacije, razvite s pomočjo `OMNIThread` abstarkcije, so lažje prenosljive med različnimi arhitekturami in osnovnimi nitnimi vmesniki.

Programski vmesnik je razvit tako, da je podoben vmesniku programskega jezika C za delo s POSIX (ang. Portable **O**perating **S**ystem **I**nterface for Uni**x;** slo. Prenosni vmesnik operacijskega sistema za Unix) nitmi. Večina implementacije je sestavljene iz C++ ovojnih funkcij okoli klicev sistemskih pthread funkcij. Nekatere kompleksnejše funkcionalnosti pthread implementacije v tej abstrakciji niso podprte, saj je težko zagotoviti enake značilnosti tudi za nitne implemetacije v drugih sistemih. [6]

Za delo z deljenimi (skupnimi) viri (ang. shared resources) se uporabljajo sinhronizacijski objekti, kot so:

- muteksi (ang. mutexes),
- številni semaforji (ang. couting semaphores),
- dogodkovni objekti (ang. event objects),
- pogojne spremenljivke (ang. condition variables),
- čakajoča časovna stikala (ang. waitable timers) in
- kritične sekcije (ang. critical sections).

Sinhronizacija med več različnimi procesi zaenkrat še ni razvita oziroma uporabljena.

2.3 Sinhronizacijski objekti

Za delo z deljenimi oziroma skupnimi viri se uporablja različne objekte, ki jim pravimo sinhronizacijski objekti. Sinhronizacijski objekti se uporabljajo za preprečevanje tveganih stanj v kritičnih sekcijah razvite kode.

Tvegano stanje (ang. race condition) je stanje oziroma trenutek, ko dve ali več niti dostopa(ta) do deljenih podatkov. Kritična sekcija je torej del naše kode, v kateri lahko pride do teh tveganih stanj, in s tem pravo mesto za uporabo sinhronizacijskih objektov.[3]

Izmed zgoraj omenjenih sinhronizacijskih objektov bom podrobneje predstavil mutekse, pogojne spremenljivke in številne semaforje.

2.3.1 Muteks

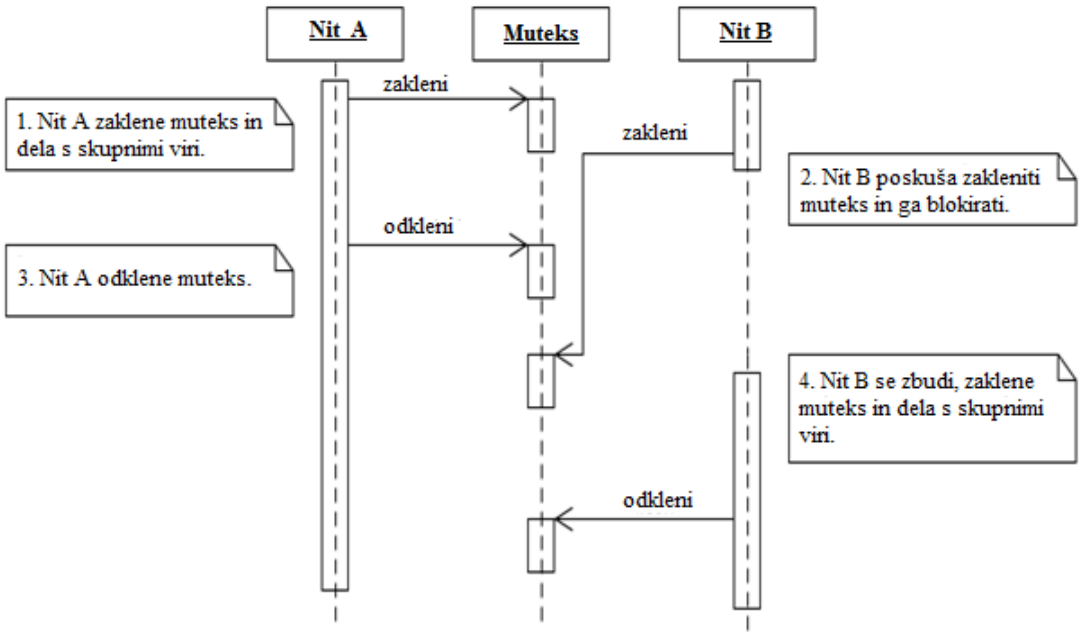
Angleška beseda mutex je okrajšava za izraz medsebojno izključevanje (ang. mutual exclusion). Kot že ime pove, se objekt muteks uporablja za vzajemno izključevanje oziroma za preprečevanje tveganih stanj pri izvajanju določenih nalog [4].

Uporabljamo ga, ko želimo preprečiti vzporedno izvajanje delov programa, ki se ne smeta izvajati v istem času, ali ko želimo preprečiti dostop do istega objekta iz več niti v istem trenutku. Ko v kritični sekciji kode uporabimo muteks, lahko do objektov v tem delu kode dostopa samo ena nit naenkrat.

Omnithread implemetacija nam omogoča dve operaciji z objektom muteks, in sicer `zakleni()` (ang. `lock()`) in `odkleni()` (ang. `unlock()`). Muteks je torej objekt, ki ga lahko nit odklene ali zaklene.

Če je recimo nit A zaklenila muteks in ga poskuša zakleniti še nit B, jo operacijski sistem ustavi in ji dovoli zakleniti muteks šele, ko ga nit A odklene. Muteks zakleni je torej mehanizem, ki prepreči dostop do podatkov iz drugih niti. Odkleni mehanizem pa dostop do podatkov spet omogoči. Muteks lahko odklene edino nit, ki ga je zaklenila.

Na sliki 5 je prikazan kontekst objekta muteks.



Slika 5: Kontekst objekta muteks [13]

2.3.2 Pogojna spremenljivka

Pogojna spremenljivka je objekt, ki se uporablja za signalizacijo med nitmi. Pove nam stanje deljenih podatkov oziroma podatkov, nad katerimi niti izvajajo naloge. Pogojno spremenljivko uporabimo takrat, ko želimo sporočiti, da se lahko nad deljenimi podatki izvede naslednja naloga, ki čaka v vrsti.

V omnithread implemetaciji nad objektom pogojna spremenkljivka uporabljamo operacije `signaliziraj()` (ang. `signal()`), `čakaj()` (ang. `wait()`) in `oddaj()` (ang. `broadcast()`).

Klic funkcije `signal()` zbudi vsaj eno nit, ki čaka na pogojno spremenljivko, če katera sploh čaka. Klic funkcije `wait()` povzroči, da nit čaka na pogojno spremenljivko, klic funkcije `broadcast()` pa zbudi vse niti, ki čakajo na pogojno spremenljivko.

Povezava med objektom pogojne spremenljivke in objektom muteksa traja, dokler obstaja pogojna spremenljivka. Isti muteks objekt se lahko uporablja tudi za več pogojnih spremenljivk.

2.3.3 Števni semafor

Števni semafor je spremenljivka, ki jo lahko povečamo do poljubne velikosti, vendar jo lahko zmanjšamo samo do ničle. Če je vrednost spremenljivke večja od nič, operacija uspe, v nasprotnem primeru pa mora nit počakati, dokler druga nit ne poveča spremenljivke.

V `OMNIThread` implementaciji nad objektom števni semafor uporabljamo funkciji `wait()` in `post()`.

Ob klicu funkcije `wait()` spremenljivko, če ni enaka nič zmanjšamo za 1, če pa je enaka nič, blokiramo izvajanje niti. Ob klicu operacije `post()` spremenljivko povečamo za 1, če ni blokirana nobena nit, če pa je katera nit blokirana, s klicem operacije `wait()` nit zbudimo.

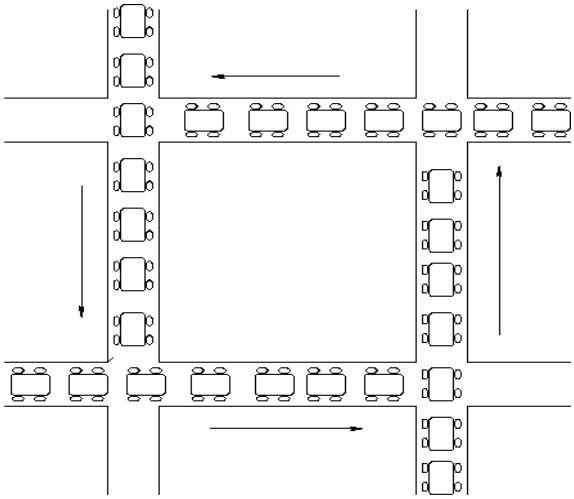
2.4 Smrtni objem

Smrtni objem (ang. deadlock) je posebno stanje programa, ko dve ali več niti čaka druge niti, da sprostijo deljene vire, ki so jih zaklenile. Ker niti čakajo ena drugo, se zaklenjeni deljeni viri (podatki) ne morejo sprostiti oziroma odkleniti. Program zato ne more nadaljevati z izvajanjem teh delov programa in je v tako imenovanem smrtnem objemu.[7]

Da lahko pride do smrtnega objema, morajo biti izpolnjeni štirje pogoji. Vsak vir je ali prost ali dodeljen eni niti. Nit, ki trenutno uporablja vir, ne more zahtevati novih virov. Ko nit enkrat uporablja vir, ga nihče ne more uporabiti. Vsaka nit čaka na vir, ki ga uporablja druga nit.

Vrsta smrtnega objema je tudi samostojni smrtni objem (ang. self deadlock). To je pojav, ko hoče ena sama nit dvakrat zakleniti muteks.[7]

Na sliki 6 je na zanimiv način prikazan smrtni objem.



Slika 6: Smrtni objem [7]

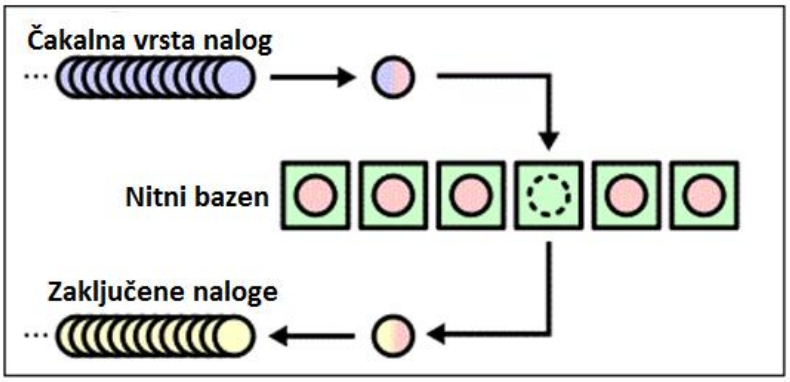
2.5 Nitni bazen

Upravljanje z vsako nitjo posebej je precej nezaželen način razvoja večnitnih aplikacij. Ustvarjanje objekta za vsako nit, nadzor nad njegovo življenjsko dobo in določitev primernega števila niti so stvari, na katere je potrebno biti pozoren pri takem načinu razvoja večnitnih aplikacij.

V veliki večini večnitnih aplikacij za upravljanje z nitmi skrbijo nitni bazeni (ang. threadpool).

Nitni bazen je torej množica, ki vsebuje skupino delovnih niti za izvajanje časovno dolgotrajnih operacij vzporedno. Naloge dodeljuje delovnim nitim (ang. worker threads) tako, da nalogo postavi v čakalno vrsto, kjer jo prva prosta nit vzame iz čakalne vrste in izvrši. Vsaka delovna nit v nitnem bazenu lahko opravi katero koli nalogo.[10]

Na sliki 7 je prikazano dodeljevanje nalog delovnim nitim.



Slika 7: Dodeljevanje nalog delovnim nitim v nitnem bazenu [10]

Najpreprostejše različice nitnih bazenov imajo stalno število delovnih niti, ki izvajajo naloge v čakalni vrsti.

Zahtevnejše različice nitnih bazenov pa imajo tako stalne (ang. persistent) kot tudi nestalne (ang. non-persistent) delovne niti. Nestalne delovne niti se kreirajo, ko je nalog v vrsti preveč in morajo le-te čakati stalne delovne niti, da opravijo svoje naloge.

Prednost nestalnih delovnih niti v nitnem bazenu je, da se kreirajo, ko jih aplikacija potrebuje. S tem dosežemo, da aplikacija obremeni operacijski sistem z novimi nitmi, šele ko je to nujno potrebno, ne pa vedno.

Če aplikacija v delovnih nitih izvaja naloge z različno prioriteto, se nove delovne niti lahko kreirajo le za naloge z najvišjo prioriteto, ne pa za vse. Da ne bi preveč obremenjevali operacijskega sistema, pri manj pomembnih nalogah ne ustvarjamo novih niti. To sicer pomeni, da bo čas izvajanja tovrstnih nalog daljši, vendar te naloge tako in tako niso zelo pomembne in se lahko izvedejo tudi kasneje.

3 Razvoj aplikacije in enostavnega nitnega bazena

Cilj diplomskega dela je razvoj enostavnega nitnega bazena in s pomočjo enostavne aplikacije za kopiranje datotek iz enega v drug imenik pokazati prednosti nitnega bazena oziroma prednosti uporabe več delovnih niti sočasno, tako stalnih kot tudi nestalnih. Samo nalogo razdelimo v dva dela.

Prvi del naloge je razvoj enostavne aplikacije. Ta del vključuje razvoj razreda za delo z nitmi, ki bo vseboval ovojne funkcije klicev `OMNIThread` funkcij, razvoj samega nitnega bazena, ki bo z nitmi upravljal in jim dodeljeval naloge, razvoj razreda za delo z delovnimi nitmi, razvoj funkcionalnosti kopiranja in pa tudi razvoj uporabniškega vmesnika v ukazni vrstici za zagon kopiranja s podanimi vhodnimi podatki, ki bodo pravilno interpretirani.

Drugi del naloge pa je izvedba testov ter analiza in primerjava rezultatov. Z različnimi vhodnimi in testnimi podatki bomo z različnim testnimi primeri pokazali in analizirali delovanje nitnega bazena in hitrost aplikacije oziroma kopiranja. Zanimal nas bo predvsem čas izvajanja celotnega kopiranja. Poleg časa izvajanja naloge kopiranja pa bomo spremljali tudi število nestalnih niti, ki so se kreirale, in količina sistemskih virov, ki jih je aplikacija uporabljala med svojim delovanjem pri največji obremenitvi.

Naša aplikacija bo kot vhodne parametre prejela največje možno število delovnih niti v nitnem bazenu, število stalnih delovnih niti v nitnem bazenu, absolutno pot do imenika, iz katerega bo datoteke kopirala, in še absolutno pot ciljnega imenika.

V primeru doseženega največjega možnega števila delovnih niti želimo, da bo naša aplikacija prenehala s kreiranjem novih nestalnih delovnih niti in preložila izvajanje naslednje operacije, dokler ne bo prosta prva delovna nit.

3.1 Omejitve pri razvoju in uporabi nitnega bazena

Pri razvoju večnitnih aplikacij moramo biti pozorni tudi na določene sistemske omejitve. Ker je naša aplikacija razvita za operacijski sistem Windows, se bomo v tem diplomskem delu omejili le na omejitve operacijskih sistemov Windows, in sicer Windows Server 2008 R2, Windows 7 64 bit (ki ima za osnovo isto platformo kot Windows Server 2008 R2) in Windows 7 32 bit. Prva dva operacijska sistema imata 64-bitno arhitekturo, medtem ko je naša aplikacija 32-bitna.

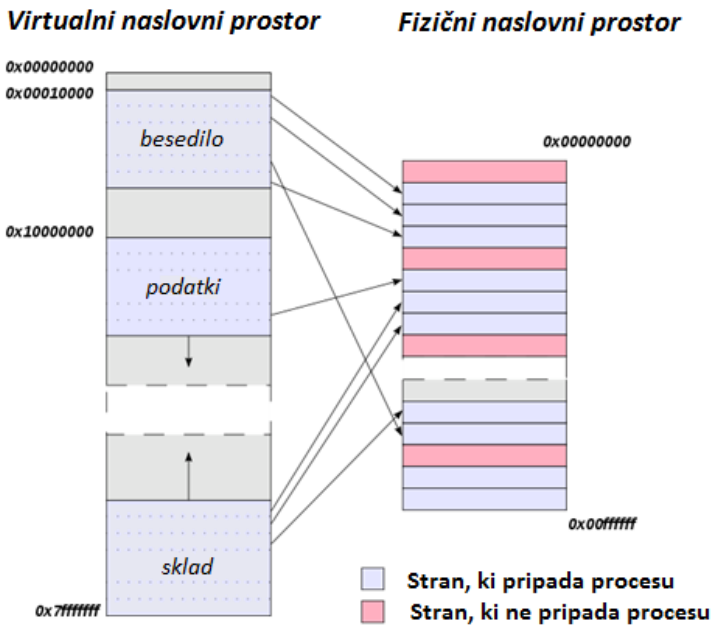
V analitičnem oziroma primerjalnem delu naloge bomo pri določitvi vhodnih parametrov in med samim izvajanjem nalog te omejitve upoštevali in jih med samim izvajanjem testov tudi spremljali. Ti podatki nam bodo pomoč pri določitvi optimalnih vhodnih podatkov za izvajanje naših nalog.

Procesi in niti za svoje delovanje uporabljajo fizični spomin (ang. physical memory), navidezni spomin (ang. virtual memory) in bazenski spomin (ang. pool memory). Število niti, ki se lahko na sistemu kreira, je odvisno od zgoraj omenjenih sistemskih virov.

Windows proces je v bistvu neka posoda, ki gosti izvedbo izvedljivih datotek oziroma slik. Predstavljen je z objektom jedra procesa, ki ga Windows skupaj z njegovimi strukturami podatkov uporablja za shranjevanje informacij in pri sledenju informacij o izvedljivi datoteki.

Proces ima virtualni naslovni prostor (ang. virtual address space), ki hrani zasebne in deljene podatke procesa. Sam proces ne izvaja nalog, ampak naloge izvajajo niti v procesu. Proces sam pa ima lahko seveda eno ali več niti.

Na sliki 8 sta prikazana tako virtualni kot fizični naslovni prostor.



Slika 8: Virtualni in fizični naslovni prostor [14]

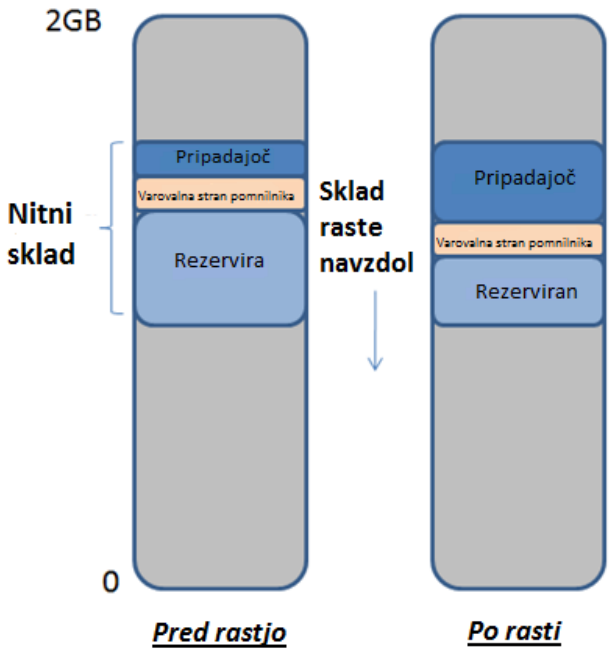
Poleg osnovnih informacij o nitih (kot so stanje CPE registra, prioritete, uporaba virov) ima vsaka nit tudi svoj del naslovnega prostora, imenovan sklad (ang. stack). Ta sklad nit lahko uporabi za vzdrževanje lokalnih spremenljivk, za podajanje parametrov funkcijam, za shranjevanje povratnih naslovov funkcij itd. Na začetku je dodeljen le del spomina sklada, ostali del pa je samo rezerviran, tako da ne porabimo celotnega spomina sklada po nepotrebnem.

32-bitni sistem po navadi dodeli 4 KB spomina, rezervira pa še dodaten 1 MB spomina za vsak sklad. Ker vsaka nit uporablja del naslovnega prostora, je tako število niti, ki jih proces lahko ustvari, omejeno na število, ki ga dobimo, če delimo velikost naslovnega prostora procesa z velikostjo sklada niti.[8]

Za primerjavo: 32-bitni sistem z 2 GB naslovnega prostora lahko kljub temu da je ves naslovni prostor na voljo samo za sklade niti, ustvari največ 2048 niti, kar pa je v praksi praktično nemogoče doseči. [8]

32-bitni proces na 64-bitnem operacijskem sistemu z 4 GB naslovnega prostora lahko ustvari največ 3204 niti. Pričakovali bi, da lahko 32-bitni proces na 64-bitnem sistemu ustvari 4096 niti, vendar temu ni tako. 32-bitni proces se na 64-bitnem sistemu izvaja kot 64-bitni proces z obnašanjem kot 32-bitna nit. Tako sta za vsako nit rezervirana 32-bitni in 64-bitni sklad. 64-bitni sklad ima rezerviranih 256 KB. Vsaka nit tako uporablja 256 KB + 1 MB naslovnega prostora.[8]

Na sliki 9 je prikazan nitni sklad.



Slika 9: Rezerviran in pripadajoč spomin v nitnem skladu[8]

Največje možno število niti se od operacijskega sistema do operacijskega sistema razlikuje. Na 64-bitnem operacijskem sistemu Windows 7 je tako lahko mogoče kreirati le 2925 niti.

Največje število procesov na sistemu je seveda manjše od skupnega števila niti. Vsak proces namreč potrebuje poleg spomina za upravljanje niti tudi nekaj spomina za svoje delovanje.

3.2 Definicija in razvoj uporabniškega vmesnika

Ena izmed manjših nalog diplomskega dela je tudi razvoj uporabniškega vmesnika, ki nam bo omogočal zagon aplikacije in podajanje potrebnih vhodnih podatkov ostalim razredom oziroma funkcionalnostim.

Sama aplikacija je sestavljena iz ene izvršljive datoteke `(.exe)`, ki se imenuje `FileCopy.exe` in ene dinamične knjižnice `(.dll)`, ki se imenuje `common.dll`.

Razvita aplikacija je konzolna aplikacija, zato bo izvršljiva datoteka `FileCopy.exe` s potrebnimi vhodnimi parametri tudi naš uporabniški vmesnik v ukazni vrstici.

Ob klicu `FileCopy.exe` datoteke bo treba v ukazni vrstici podati tudi štiri obvezne vhodne parametre.

Ti parametri so po vrstnem redu:

- največje želeno število delovnih niti v nitnem bazenu,
- število stalnih delovnih niti v nitnem bazenu,
- absolutna pot začetnega imenika in
- absolutna pot končnega imenika.

Na spodnji sliki je prikazan primer uporabe uporabniškega vmesnika v ukazni vrstici.

```
l:\Diploma\FileCopyApp\Release>FileCopy.exe
Usage:
FileCopy MaxNumOfThreads NumOfPersistThreads SourceDir DestinationDir

l:\Diploma\FileCopyApp\Release>FileCopy.exe 20 5 c:\ZacetniDir c:\KoncniDir
```

Slika 10: Uporabniški vmesnik

Edina naloga izvršljive datoteke je, da pravilno interpretira podane vhodne parametre, ustvari objekt nitnega bazena in mu poda pravilno interpretirane dobljene vhodne parametre.

3.3 Definicija in razvoj razreda za delo z nitmi

V dinamični knjižnici so razviti preostali potrebni razredi in funkcionalnosti.

Prvi razred, ki smo ga razvili, je razred `Thread`. Razred `Thread` zagotavlja funkcionalnost za delo z nitmi. Drugi razred, ki smo ga razvili, je razred `Mutex`, tretji `MutexLock` in še četri razred `Condition`. Ti razredi zagotavljajo funkcionalnost sinhronizacijskih objektov, so razviti v isti datoteki kot razred `Thread`, in sicer `thread.cpp`. Deklaracije teh razredov in njihovih funkcij pa so shranjene v datoteki `thread.h`.

Funkcije, razvite v teh razredih, so v glavnem samo ovojne funkcije okoli klicev sistemskih `OMNIThread` funkcij.

V datoteki `thread.h` smo datoteko `omnithread.h`, v kateri so definirane sistemske `OMNIThread` funkcije, tudi vključili.

Potrebne `omniORB` datoteke oziroma `OMNIThread` abstakcijo, ki je brezplačno na voljo pod pogoji licence GNU (ang. Lesser General Public License), smo prenesli z vira.[15]. Datoteke smo razširili v svojo mapo v projektnem imeniku.

V spodnjih vrsticah je prikazana datoteka `thread.h`, v kateri so deklarirani zgoraj omenjeni razredi in njihove funkcije.
Prvi dve vrstici prikazujeta vključevanje potrebnih datotek (`common.h` in `omnithread.h`).

```
#include "common.h"
#include <omnithread.h>
```

V razredu `Mutex` so sta poleg `konstruktorja` in `destruktorja` razviti še funkciji za zaklepanje (`Lock()`) in odklepanje (`Unlock()`) objekta muteks.

```
class DllExport Mutex : public omni_mutex {
public:
    Mutex() : omni_mutex() {};
    virtual ~Mutex() {};
    void Lock();
    void Unlock();
};
```

V razredu `MutexLock` sta razvita `konstruktor` in `destruktor`, ki avtomatsko zakleneta in odkleneta muteks.

```
class DllExport MutexLock {
    Mutex& m_mutex;
public:
    MutexLock(Mutex& a_mutex);

    ~MutexLock() {
        m_mutex.Unlock();
    };
private:
    MutexLock(const MutexLock&);
    MutexLock& operator=(const MutexLock&);
};
```

V razredu `Condition` so poleg `konstruktorja` in `destruktorja` razvite še funkcije `Čakaj()` (`Wait()`), `Signaliziraj()` (`Signal()`) in `Oddaj()` (`Broadcast()`), ki niti sporočajo, ali se lahko izvaja ali ne.

```
class DllExport Condition : public omni_condition {
public:
    Condition(Mutex* a_mutex) : omni_condition(a_mutex) {};
    void Wait();
    int TimedWait(unsigned long secs,
                  unsigned long nanosecs = 0);
    void Signal();
    void Broadcast();
};
```

V razredu `Thread` so poleg `konstruktorja` razvite še funkcije `Start()`, `StartUndetached()`, `Run()`, `RunUndetached()`, ki omogočajo zagon stalnih in nestalnih niti, funkcija `Join()`, ki povzroči, da nit, ki je klicala to funkcijo, počaka drugo nit, da se konča.

```
class DllExport Thread : public omni_thread {
```

```
public:

    Thread(): omni_thread(NULL, omni_thread::PRIORITY_NORMAL){
        // Empty
    };

    void Start();

    void StartUndetached();

    void Join();

private:

    virtual void run(void *arg);
    virtual void* run_undetached(void* arg);

    virtual void Run(void* arg) {};
    virtual void* RunUndetached(void* arg) { return NULL; }
};
```

3.4 Definicija in razvoj razreda nitnega bazena

`Threadpool` razred, ki ustvarja delovne niti in jim dodeljuje naloge, je prav tako razvit v tej knjižnici.

Naloga razvitega razreda nitnega bazena je, da ob klicu funkcije `Start()` ustvari stalne delovne niti, da razporeja delo prostim delovnimi nitim glede na število nalog oziroma operacij, ki jih mora aplikacija opraviti in da po potrebi ustvari nove nestalne delovne niti.

Razred `Threadpool` ob klicu funkcije `Start()` prejme dva vhodna parametra. Prvi vhodni parameter je parameter, ki določala največje želeno število delovnih niti v nitnem bazenu. Ta parameter nitni bazen upošteva, ko ustvarja nove nestalne delovne niti, saj skupno število delovnih niti (stalnih in nestalnih) ne sme preseči te vrednosti.

Drugi parameter določa število stalnih delovnih niti v nitnem bazenu.

V spodnjih vrsticah je prikazana datoteka `threadpool.h`, v kateri so deklarirane funkcije razreda `ThreadPool`.

Za delo z delovnimi nitmi je treba vključiti datoteko `workerthread.h`, v kateri so deklarirane funkcije razreda `WorkerThread`, ki je opisan v kasnejšem besedilu.

```
#include "workerthread.h"
```

V razredu `ThreadPool` sta poleg konstruktorja razviti funkciji `Start()` in `Stop()`, ki v nitnem bazenu skrbita za ustvarjanje in uničevanje delovnih niti.

```
class DllExport ThreadPool {

public:

    ThreadPool();

    void Start(   const unsigned long a_maxNumOfThreads,
                  const unsigned long a_numPersist);

    void Stop();
```

Funkcija, ki v razredu `ThreadPool` skrbi za dodeljevanje nalog delovnim nitim, je funkcija `Assign()`. Funkcija kot vhodni parameter prejme kazalec na nalogo, ki jo mora izvesti, in to nalogo preda prosti delovni niti.

```
     void Assign(Task* a_task_p);
```

V spodnjih vrsticah je prikazana definicija funkcije `Assign()`.

```
void ThreadPool::Assign(Task* a_task_p) {
    unsigned long numOfPThreads(0);
    {
        MutexLock l(m_persisThreadsLock_x);
        numOfPThreads = m_persisThreads_v.size();
        for (unsigned long i(0); i < m_persisThreads_v.size(); i++) {
           if (m_persisThreads_v[i]->Idle()) {
               m_persisThreads_v[i]->Assign(a_task_p);
               return;
           }
        }
    }
    MutexLock lock(m_nonPersisThreadsLock_x);
    for (unsigned long i(0); i < m_nonPersisThreads_v.size(); i++) {
       if (m_nonPersisThreads_v[i]->Idle()) {
           m_nonPersisThreads_v[i]->Assign(a_task_p);
           return;
       }
    }
    if(m_nonPersisThreads_v.size() < (m_maxNumOfThreads - numOfPThreads)) {
        // Creating a non persistent thead;
        WorkerThread* wt_p = new WorkerThread(this);
        wt_p->Start();
        m_nonPersisThreads_v.push_back(wt_p);
        // Waiting for a non persist thread to start
        Sleep(1000);
        wt_p->Assign(a_task_p);
        return;
    }
    else {
        throw exception("max num of nonpersist threads exceeded.");
    }
}
```

Funkciji `IsAnyRunning()` in `Delete(WorkerThread* a_workerthread_p)` služita za preverjanje, ali katera delovna nit še opravlja kakšno nalogo oziroma za brisanje nestalnih delovnih niti iz nitnega bazena.

```
    bool IsAnyRunning();

    void Delete(WorkerThread* a_workerthread_p);
```

Funkciji `GetNumOfCopied()` in `IncreaseNumOfCopied()` sta funkciji, ki vračata število prekopiranih datotek oziroma to število po uspešno končanem kopiranju povečata.

```
    unsigned long GetNumOfCopied();

    void IncreaseNumOfCopied();

private:

    unsigned long m_maxNumOfThreads;

    unsigned long m_numofcopied;
    Mutex         m_numOfCopied_x;

    vector<WorkerThread*> m_persisThreads_v;
    Mutex                 m_persisThreadsLock_x;

    vector<WorkerThread*> m_nonPersisThreads_v;
    Mutex                 m_nonPersisThreadsLock_x;
};

    vector<WorkerThread*> m_nonPersisThreads_v;
    Mutex                 m_nonPersisThreadsLock_x;
};
```

3.5 Definicija in razvoj razreda za delo z delovnimi nitmi

Poseben razred, imenovan `WorkerThread`, je razvit za delo z delovnimi nitmi. V tem razredu so implementirane funkcije, ki naloge, dodeljene delovnim nitim, izvedejo. Večina funkcij in spremenljivk v tem razredu se uporablja za pravilno delovanje, ustvarjanje in ustavljanje same delovne niti.

V spodnjih vrsticah je prikazana datoteka `workerthread.h`, v kateri so deklarirane funkcije za delo z delovnimi nitmi.

Poleg `konstruktorja` in `destruktorja` so razvite še funkcija `Assign()`, ki dodeli prejeto nalogo, funkcija `Stop()`, ki delovno nit ustavi, in funkcija `Idle()`, ki nam pove, ali nit miruje ali opravlja kakšno nalogo.

```
#include "thread.h"

class ThreadPool;
class Task;

class WorkerThread : public Thread {

public:
    WorkerThread(ThreadPool* a_threadpool_p);
    ~WorkerThread();
```

```
    void Stop();
    void Assign(Task* a_task_p);

    bool Idle();
private:

    ThreadPool* m_threadpool_p;

    // run non persist threads
    virtual void  Run(void* arg);
    // run persist threads
    virtual void* RunUndetached(void* arg);

    bool        m_stopped;

    Task*       m_task_p;
    Mutex       m_task_x;
    Condition   m_task_c;

    Mutex       m_idle_x;
    bool        m_idle;
};
```

3.6 Definicija in razvoj funkcionalnosti kopiranja

Sama funkcionalnost kopiranja je razvita v izvršljivi datoteki `FileCopy.exe` v razredu `CopyTask`. V tem razredu je poleg konstruktorja, ki nalogi nastavi začetni in končni imenik, razvita še funkcija `Execute()`, ki samo kopiranje izvede.

V spodnjih vrsticah je prikazana datoteka `copytask.h`, v kateri se deklarirane funkcije razreda `CopyTask`.

Vključili smo datoteko `task.h`, kjer so deklaracije nadrazreda.

```
#include "task.h"

class Mutex;

class CopyTask : public Task {

public:
    CopyTask(   const string& a_sourceFile,
                const string& a_destFile,
                ThreadPool& a_threadpool);

    ~CopyTask();

    void Execute();

private:

    ThreadPool& m_threadpool;

    const string m_sourceFile;
```

```
    const string m_destFile;
};
```

3.7 Potek delovanja aplikacije

Razvita aplikacija je konzolna aplikacija, ki se požene iz ukazne vrstice. V ukazni vrstici se kot vhodni parametri podajo največje želeno število delovnih niti, število stalnih delovnih niti, absolutna pod do začetnega imenika in absolutna pot končnega imenika.

V `main()` funkciji se podani vhodni parametri preberejo, interpretirajo. Prva parametra se podata funkciji `Start()` objektu razreda `Threadpool`, ki se kreira. V funkciji `Start()` se kreirajo stalne delovne niti.

V nadaljevanju izvajanja aplikacije se prebere vsebina začetnega imenika. Ko je vsebina prebrana, se za vsako datoteko ustvari naloga objekta `CopyTask`, ki se kot parameter poda funkciji `Assign()` objekta `ThreadPool`.

Naloga se potem v nadaljevanju dodeli prvi prosti delovni niti v nitnem bazenu. Če so stalne delovne niti proste, se dodeli stalnim delovnim nitim. V nasprotnem primeru pa se naloga dodeli prvi prosti nestalni delovni niti. Če pa ni prosta nobena nestalna delovna nit, se lahko ustvari nova delovna nit, če največje želeno število delovnih niti še ni doseženo.

Ko so vse datoteke prekopirane, se počaka, da se vse niti uničijo, in aplikacija se zaključi.

4 Testiranje razvite aplikacije in nitnega bazena

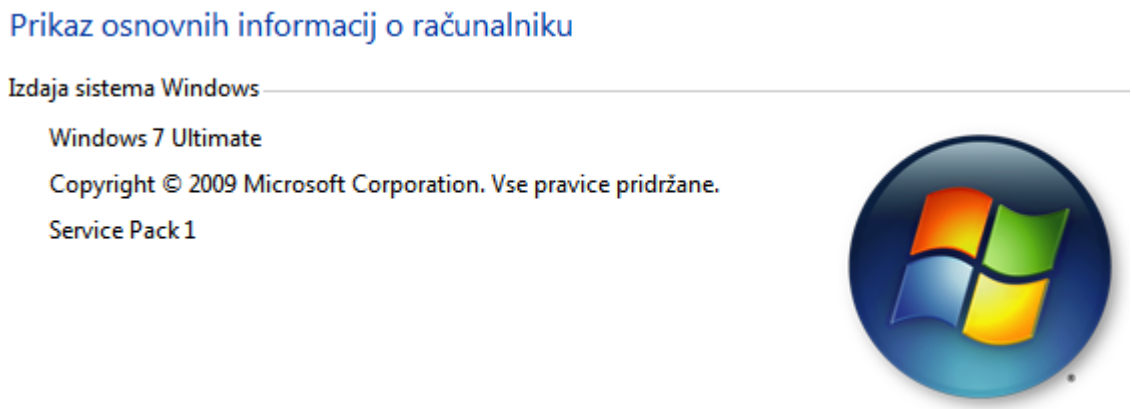
Drugi del diplomskega dela predstavlja testiranje razvite aplikacije z različnimi vrednostmi vhodnih podatkov (števila stalnih in nestalnih delovnih niti in velikosti datotek za kopiranje). Za prikazanimi rezultati testov so predstavljene ugotovitve, ki smo jih dobili z analizo in primerjavo rezultatov.

Razvita aplikacija je bila testirana na dveh operacijskih sistemih. Na vsakem sistemu je bilo opravljenih dvanajst testov s tremi različnimi kombinacijami datotek in štirimi različnimi nitnimi bazeni.

4.1 Testno okolje

Testiranje razvite aplikacije in nitnega bazena bomo izvajali tako na 32-bitnem kot tudi na 64-bitnem operacijskem sistemu Windows 7 z nameščenim servisnim paketom ena.

Na sliki 10 so predstavljeni osnovni podatki operacijskega sistema.



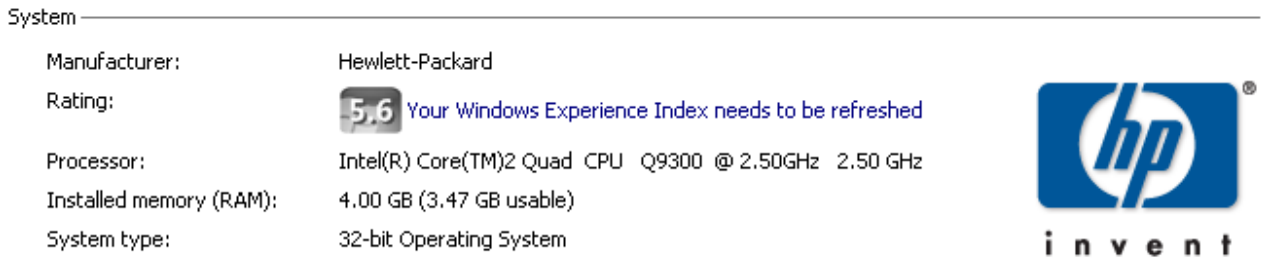
Slika 11: Podatki o verziji operacijskega sistema

Oba operacijska sistema sta nameščena na isti strojni opremi. Za nas sta najpomembnejša podatka o strojni opremi moč procesorja in količina delovnega spomina.

Moč procesorja je seveda na obeh sistemih enaka, in sicer je v računalniku vgrajen Intelov štirijedrni procesor Core2 Quad s frekvenco 2,50 GHz.

Količina delovnega spomina pa se na sistemih razlikuje. Zaradi svoje arhitekture 32-bitni operacijski sistem lahko od razpoložljivih 4GB uporablja samo 3,47 GB delovnega spomina. 64-bitni operacijski sistem pa seveda lahko uporablja ves razpoložljiv delovni spomin.

Na sliki 11 je razvidna uporaba delovnega spomina na 32-bitnem operacijskem sistemu.



Slika 12: Uporaba delovnega spomina na 32-bitnem operacijskem sistemu

Na obeh operacijskih sistemih bomo izvajali iste testne primere, to je kopiranje istih datotek iz istih začetnih imenikov v iste ciljne imenike. Tako bomo lahko tudi primerjali obnašanje aplikacije in razvitega nitnega bazena na 32-bitnem in 64-bitnem operacijskem sistemu, kar pa seveda ni glavna naloga tega diplomskega dela, ampak samo dodatna zanimivost.

Za naše testne namene bomo na zunanjem disku, priključenem na USB vrata, kreirali več začetnih in več ciljnih imenikov. V začetnih imenikih bomo ustvarili večje število različno velikih datotek, ki jih bo morala naša razvita aplikacija prekopirati v ciljne imenike.

Testni primeri so podrobneje opisani v naslednjem poglavju.

4.2 Testni primeri

Med izvajanem testov nas je zanimalo več podatkov, ki smo jih potem interpretirali in analizirali.

Predvidevali smo, da bo aplikacija uporabljala več delovnega spomina in več procesorja v testnih primerih, kjer se bodo kopirale večje datoteke, in v testih, kjer bo nitni bazen lahko ustvaril in uporabljal večje število delovnih niti.

Najpomembnejši podatek testiranja je celoten čas izvajanja ene operacije kopiranja. V ta čas štejemo čas, potreben za kreiranje nitnega bazena, stalnih in nestalnih delovnih niti v nitnem bazenu ter čas samega kopiranja datotek.

Drugi pomemben podatek je število nestalnih niti, ki so se med izvajanem testa ustvarile v nitnem bazenu in izvajale operacije kopiranja datotek.

Tretji in četrti podatek sta zelo povezana. Količina delovnega spomina in procesorja, ki ga je aplikacija (oziroma njene delovne niti) med izvajanem kopiranja uporabljala, sta podatka, ki sta neposredno povezana z razvojem večnitnih aplikacij, in ju je potrebno vsekakor upoštevati.

Za potrebe analize in primerjave rezultatov so bili izvedeni trije različni testni primeri. Za vsak testni primer smo izvedli štiri teste, v katerih so bili uporabljeni različni nitni bazeni, vsak test je bil ponovljen dvakrat.

Razlika med testnimi primeri je v tem, da se je v prvem testnem primeru izvajalo kopiranje manjših datotek velikosti približno 1KB, v drugem testnem primeru se je kopirala polovica manjših in polovica večjih datotek velikosti približno 1MB, v tretjem testnem primeru pa se je kopiralo samo večje datoteke.

Sami testni se med seboj razlikujejo po operacijskem sistemu, na katerem so bili izvedeni, in po številu stalnih in nestalnih delovnih niti, ki so lahko na voljo nitnemu bazenu za izvajanje operacij kopiranja.

V vseh testnih primerih je bil prvi test izveden samo z eno stalno delovno nitjo v nitnem bazenu. Rezultati tega testa služijo kot osnova za primerjavo, koliko dodatne stalne in nestalne delovne niti v nitnem bazenu povečajo učinkovitost aplikacije oziroma izboljšajo hitrost kopiranja datotek iz enega v drugi imenik.

Testni primer 1

V začetnem imeniku je 10000 tekstovnih datotek velikosti približno 1KB, ki jih je treba skopirati v ciljni imenik.

V testih 1 in 2 z uporabo ene stalne delovne niti v nitnem bazenu skopiramo datoteke iz začetnega v ciljni imenik. Testa se med seboj razlikujeta po operacijskem sistemu.

V testih 3 in 4 z uporabo petih stalnih in petih nestalnih delovnih niti v nitnem bazenu skopiramo datoteke iz začetnega v ciljni imenik. Testa se med seboj razlikujeta po operacijskem sistemu.

V testih 5 in 6 z uporabo petih stalnih in petindvajsetih nestalnih delovnih niti v nitnem bazenu skopiramo datoteke iz začetnega v ciljni imenik. Testa se med seboj razlikujeta po operacijskem sistemu.

V testih 7 in 8 z uporabo desetih stalnih in devetdesetih nestalnih delovnih niti v nitnem bazenu skopiramo datoteke iz začetnega v ciljni imenik. Testa se med seboj razlikujeta po operacijskem sistemu.

V tabeli 1 so prikazani testi testnega primera 1.

TESTNI PRIMER	TEST	ARHITEKTURA OS	ŠT. MANJŠIH DATOTEK V IMENIKU	ŠT. VEČJIH DATOTEK V IMENIKU	ŠT. STALNIH DELOVNIH NITI	ŠT. NESTALNIH DELOVNIH NITI
1	1	32	10000	0	1	0
	2	64			1	0
	3	32			5	5
	4	64			5	5
	5	32			5	25
	6	64			5	25
	7	32			10	90
	8	64			10	90

Tabela 1: Testni primer 1

Testni primer 2

V začetnem imeniku je 5000 manjših tekstovnih datotek velikosti približno 1KB in 5000 večjih datotek velikosti približno 1MB, ki jih je treba skopirati v ciljni imenik.

V testih 9 in 10 z uporabo ene stalne delovne niti v nitnem bazenu skopiramo datoteke iz začetnega v ciljni imenik. Testa se med seboj razlikujeta po operacijskem sistemu.

V testih 11 in 12 z uporabo petih stalnih in petih nestalnih delovnih niti v nitnem bazenu skopiramo datoteke iz začetnega v ciljni imenik. Testa se med seboj razlikujeta po operacijskem sistemu.

V testih 13 in 14 z uporabo petih stalnih in petindvajsetih nestalnih delovnih niti v nitnem bazenu skopiramo datoteke iz začetnega v ciljni imenik. Testa se med seboj razlikujeta po operacijskem sistemu.

V testih 15 in 16 z uporabo desetih stalnih in devetdesetih nestalnih delovnih niti v nitnem bazenu skopiramo datoteke iz začetnega v ciljni imenik. Testa se med seboj razlikujeta po operacijskem sistemu.

V tabeli 2 so prikazani testi testnega primera 2.

TESTNI PRIMER	TEST	ARHITEKTURA OS	ŠT. MANJŠIH DATOTEK V IMENIKU	ŠT. VEČJIH DATOTEK V IMENIKU	ŠT. STALNIH DELOVNIH NITI	ŠT. NESTALNIH DELOVNIH NITI
2	9	32	5000	5000	1	0
	10	64			1	0
	11	32			5	5
	12	64			5	5
	13	32			5	25
	14	64			5	25
	15	32			10	90
	16	64			10	90

Tabela 2: Testni primer 2

Testni primer 3

V začetnem imeniku je 10000 večjih datotek velikosti približno 1MB, ki jih je treba skopirati v ciljni imenik.

V testih 17 in 18 z uporabo ene stalne delovne niti v nitnem bazenu skopiramo datoteke iz začetnega v ciljni imenik. Testa se med seboj razlikujeta po operacijskem sistemu.

V testih 19 in 20 z uporabo petih stalnih in petih nestalnih delovnih niti v nitnem bazenu skopiramo datoteke iz začetnega v ciljni imenik. Testa se med seboj razlikujeta po operacijskem sistemu.

V testih 21 in 22 z uporabo petih stalnih in petindvajsetih nestalnih delovnih niti v nitnem bazenu skopiramo datoteke iz začetnega v ciljni imenik. Testa se med seboj razlikujeta po operacijskem sistemu.

V testih 23 in 24 z uporabo desetih stalnih in devetdesetih nestalnih delovnih niti v nitnem bazenu skopiramo datoteke iz začetnega v ciljni imenik. Testa se med seboj razlikujeta po operacijskem sistemu.

V tabeli 3 so prikazani testi testnega primera 3.

TESTNI PRIMER	TEST	ARHITEKTURA OS	ŠT. MANJŠIH DATOTEK V IMENIKU	ŠT. VEČJIH DATOTEK V IMENIKU	ŠT. STALNIH DELOVNIH NITI	ŠT. NESTALNIH DELOVNIH NITI
3	17	32	0	10000	1	0
	18	64			1	0
	19	32			5	5
	20	64			5	5
	21	32			5	25
	22	64			5	25
	23	32			10	90
	24	64			10	90

Tabela 3: Testni primer 3

5 Rezultati in analiza

Rezultati testov so predstavljeni v tabeli in tudi opisani. Vsi testi v vseh testnih primerih so bili izvedeni na 32- in 64-bitnem operacijskem sistemu. Vsak test je bil izveden dvakrat, s čimer smo se prepričali, da so podatki pravilni oziroma rezultati testa niso naključni.

5.1 Rezultati izvedenih testov

Testni primer 1

V testnem primeru 1 smo kopirali manjše datoteke velikosti približno 1 KB iz začetnega v končni imenik. Kopiranje manjših datotek je zelo hitra operacija, kar je tudi razvidno iz rezultatov.

V testih 1 in 2 smo vse datoteke prekopirali z uporabo ene same stalne delovne niti, kar je enako, kot da bi aplikacija delovala brez nitnega bazena.

Kopiranje je na 64-bitnem sistemu trajalo 145,5 sekunde, aplikacija pa je med kopiranjem uporabljala največ 4100 KB delovnega spomina.

Na 32-bitnem sistemu je kopiranje trajalo 147 sekund, aplikacija pa je med kopiranjem uporabljala največ 2500 KB delovnega spomina.

V testih 3 in 4 smo vse datoteke prekopirali z uporabo 5 stalnih delovnih niti in 5 nestalnih delovnih niti. Med kopiranjem so se ustvarile vse možne nestalne delovne niti.

Kopiranje je na 64-bitnem sistemu trajalo 93 sekund, aplikacija pa je med kopiranjem uporabljala največ 4600 KB delovnega spomina.

Na 32-bitnem sistemu je kopiranje trajalo 74,5 sekunde, aplikacija pa je med kopiranjem uporabljala največ 3150 KB delovnega spomina.

V testih 5 in 6 smo vse datoteke prekopirali z uporabo 5 stalnih in 25 nestalnih delovnih niti. Med kopiranjem so se ustvarile vse možne nestalne delovne niti.

Kopiranje je na 64-bitnem sistemu trajalo 109,5 sekunde, aplikacija pa je med kopiranjem uporabljala največ 5800 KB delovnega spomina.

Na 32-bitnem sistemu je kopiranje trajalo 98,5 sekunde, aplikacija pa je med kopiranjem uporabljala največ 3450 KB delovnega spomina.

V testih 7 in 8 smo vse datoteke prekopirali z uporabo 10 stalnih in 90 nestalnih delovnih niti. Med kopiranjem so se ustvarile vse možne nestalne delovne niti.

Kopiranje je na 64-bitnem sistemu trajalo 138,5 sekunde, aplikacija pa je med kopiranjem uporabljala največ 8300 KB delovnega spomina.

Na 32-bitnem sistemu je kopiranje trajalo 131 sekund, aplikacija pa je med kopiranjem uporabljala največ 4900 KB delovnega spomina.

V tabeli 4 so prikazani vsi rezultati testov testnega primera 1.

Testni primer	Test	OS	Stalne niti/Nestalne niti	Čas 1 (sekund)	Čas 2 (sekund)	Povprečni čas (sekund)	Največja poraba delovnega spomina (KB)
1	1	32 bit	1/0	152	142	147	~2500
	2	64 bit	1/0	146	145	145,5	~4100
	3	32 bit	5/5	73	76	74,5	~3150
	4	64 bit	5/5	92	94	93	~4600
	5	32 bit	5/25	99	98	98,5	~3450
	6	64 bit	5/25	109	110	109,5	~5800
	7	32 bit	10/90	131	131	131	~4900
	8	64 bit	10/90	139	138	138,5	~8300

Tabela 4: Rezultati testov testnega primera 1

Testni primer 2

V testnem primeru 2 smo kopirali 5000 manjših datotek velikosti približno 1 KB in 5000 malo večjih datotek velikosti 1MB iz začetnega v končni imenik. Kopiranje malo večjih datotek je počasnejša operacija, kar je bilo tudi razvidno iz testa, saj se je del testa, ki je kopiral večje datoteke, izvajal počasneje.

V testih 9 in 10 smo vse datoteke prekopirali z uporabo ene same stalne delovne niti, kar je enako, kot da bi aplikacija delovala brez nitnega bazena.

Kopiranje je na 64-bitnem sistemu trajalo 1045 sekund, aplikacija pa je med kopiranjem uporabljala največ 4200 KB delovnega spomina.

Na 32-bitnem sistemu je kopiranje trajalo 907 sekund, aplikacija pa je med kopiranjem uporabljala največ 2800 KB delovnega spomina.

V testih 11 in 12 smo vse datoteke prekopirali z uporabo 5 stalnih delovnih niti in 5 nestalnih delovnih niti. Med kopiranjem so se torej ustvarile vse možne nestalne delovne niti.

Kopiranje je na 64-bitnem sistemu trajalo 836 sekund, aplikacija pa je med kopiranjem uporabljala največ 6800 KB delovnega spomina.

Na 32-bitnem sistemu je kopiranje trajalo 890,5 sekund, aplikacija pa je med kopiranjem uporabljala največ 5600 KB delovnega spomina.

V testih 13 in 14 smo vse datoteke prekopirali z uporabo 5 stalnih in 25 nestalnih delovnih niti. Med kopiranjem so se ustvarile vse možne nestalne delovne niti.

Kopiranje je na 64-bitnem sistemu trajalo 800 sekund, aplikacija pa je med kopiranjem uporabljala največ 12600 KB delovnega spomina.

Na 32-bitnem sistemu je kopiranje trajalo 889 sekund, aplikacija pa je med kopiranjem uporabljala največ 11100 KB delovnega spomina.

V testih 15 in 16 smo vse datoteke prekopirali z uporabo 10 stalnih in 90 nestalnih delovnih niti. Med kopiranjem so se ustvarile vse možne nestalne delovne niti.

Kopiranje je na 64-bitnem sistemu trajalo 755 sekund, aplikacija pa je med kopiranjem uporabljala največ 34000 KB delovnega spomina.

Na 32-bitnem sistemu je kopiranje trajalo 711 sekund, aplikacija pa je med kopiranjem uporabljala največ 24000 KB delovnega spomina.

V tabeli 5 so prikazani vsi rezultati testov testnega primera 2.

Testni primer	Test	OS	Stalne niti/Nestalne niti	Čas 1 (sekund)	Čas 2 (sekund)	Povprečni čas (sekund)	Največja poraba delovnega spomina (KB)
2	9	32 bit	1/0	902	912	907	~2800
	10	64 bit	1/0	1040	1050	1045	~4200
	11	32 bit	5/5	893	888	890,5	~5600
	12	64 bit	5/5	825	847	836	~6800
	13	32 bit	5/25	888	890	889	~11100
	14	64 bit	5/25	805	795	800	~12600
	15	32 bit	10/90	715	707	711	~24000
	16	64 bit	10/90	757	753	755	~34000

Tabela 5: Rezultati testov testnega primera 2

Testni primer 3

V testnem primeru 1 smo kopirali 10000 malo večjih datotek velikosti 1MB iz začetnega v končni imenik. Kopiranje malo večjih datotek je počasnejša operacija, kar je bilo tudi razvidno iz testov.

V testih 17 in 18 smo vse datoteke prekopirali z uporabo ene same stalne delovne niti, kar je enako, kot da bi aplikacija delovala brez nitnega bazena.

Kopiranje je na 64-bitnem sistemu trajalo 1374 sekund, aplikacija pa je med kopiranjem uporabljala največ 4200 KB delovnega spomina.

Na 32-bitnem sistemu je kopiranje trajalo 1452,5 sekund, aplikacija pa je med kopiranjem uporabljala največ 2800 KB delovnega spomina.

V testih 19 in 20 smo vse datoteke prekopirali z uporabo 5 stalnih delovnih niti in 5 nestalnih delovnih niti. Med kopiranjem so se torej ustvarile vse možne nestalne delovne niti.

Kopiranje je trajalo na 64-bitnem sistemu trajalo 1280,5 sekunde, aplikacija pa je med kopiranjem uporabljala največ 6800 KB delovnega spomina.

Na 32-bitnem sistemu je kopiranje trajalo 1506 sekund, aplikacija pa je med kopiranjem porabljala največ 5600 KB delovnega spomina

V testih 21 in 22 smo vse datoteke prekopirali z uporabo 5 stalnih in 25 nestalnih delovnih niti. Med kopiranjem so se ustvarile vse možne nestalne delovne niti.

Kopiranje je na 64-bitnem sistemu trajalo 1294 sekund, aplikacija pa je med kopiranjem uporabljala največ 12600 KB delovnega spomina.

Na 32 bitnem sistemu je kopiranje trajalo 1487 sekund, aplikacija pa je med kopiranjem porabljala največ 11100 KB delovnega spomina.

V testih 23 in 24 smo vse datoteke prekopirali z uporabo 10 stalnih in 90 nestalnih delovnih niti. Med kopiranjem so se ustvarile vse možne nestalne delovne niti.

Kopiranje je na 64-bitnem sistemu trajalo 1210 sekund, aplikacija pa je med kopiranjem uporabljala največ 34100 KB delovnega spomina.

Na 32-bitnem sistemu je kopiranje trajalo 1133 sekund, aplikacija pa je med kopiranjem porabljala največ 27300 KB delovnega spomina.

V tabeli 6 so prikazani vsi rezultati testov testnega primera 3.

Testni primer	Test	OS	Stalne niti/Nestalne niti	Čas 1 (sekund)	Čas 2 (sekund)	Povprečni čas (sekund)	Največja poraba delovnega spomina (KB)
3	17	32 bit	1/0	1525	1380	1452,5	~2800
	18	64 bit	1/0	1370	1378	1374	~4200
	19	32 bit	5/5	1539	1473	1506	~5500
	20	64 bit	5/5	1287	1274	1280,5	~6800
	21	32 bit	5/25	1498	1476	1487	~11100
	22	64 bit	5/25	1295	1293	1294	~12600
	23	32 bit	10/90	1140	1126	1133	~27300
	24	64 bit	10/90	1216	1204	1210	~34100

Tabela 6: Rezultati testov testnega primera 3

5.2 Analiza razvite aplikacije in nitnega bazena

Rezultati testov so dokaj zanimivi in celo malo nepričakovani. Pri kopiranju manjših datotek so rezultati testov pričakovani, pri kopiranju večjih datotek pa malce nepričakovani.

Ugotovljeno pa je bilo, da je dokaj veliko omejitev pri zmogljivosti in učinkovitosti naše aplikacije, natančneje pri kopiranju večjih datotek, predstavljal trdi disk oziroma njegova zmogljivost. Kopiranje se je izvajalo na zunanjem trdem disku, priključenem na USB vrata.

Testni primer 1

V testnem primeru 1 je bilo ugotovljeno, da je kopiranje manjših datotek zelo hitra operacija.

Boljši rezultati so bili doseženi na 32-bitnem operacijskem sistemu, razen v testu s samo eno delovno nitjo, kjer je bil dosežen boljši rezultat na 64-bitnem operacijskem sistemu.

Rezultati posameznih testov so glede na število delovnih niti na obeh operacijskih sistemih primerljivi, saj je vrstni red enak.

Prednost kopiranja manjših datotek na 32-bitnem operacijskem sistemu je tudi manjša poraba delovnega spomina.

Najboljši rezultat kopiranja je bil dosežen na 32-bitnem operacijskem z uporabo 5 stalnih in 5 nestalnih delovnih niti v nitnem bazenu.

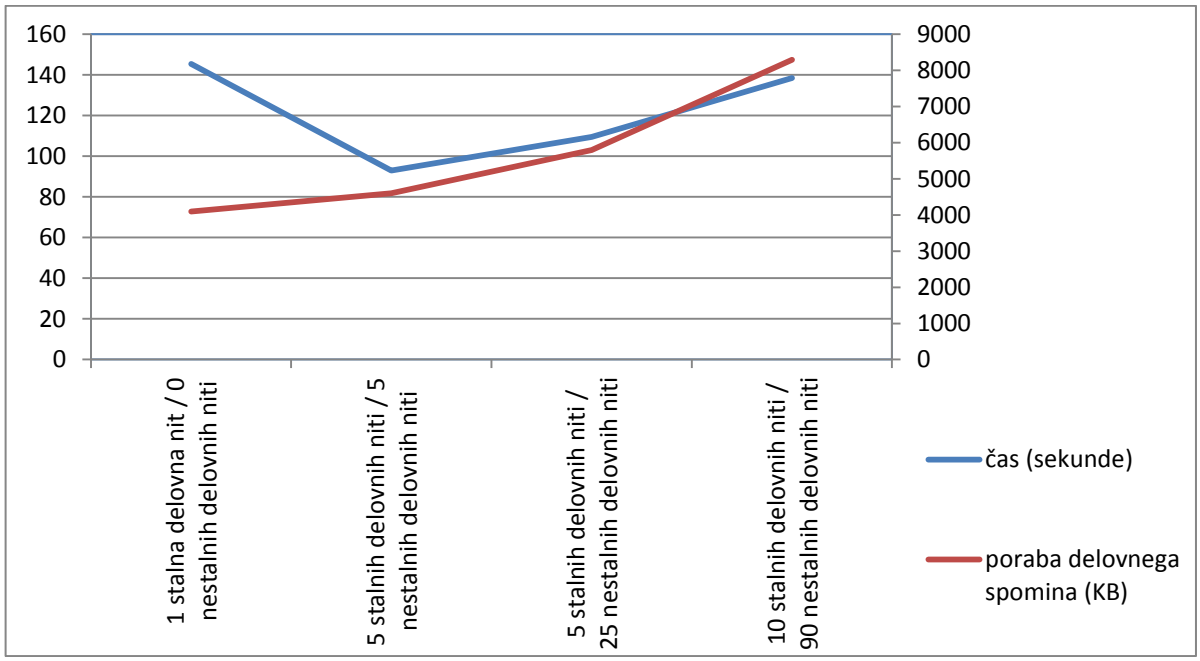
Najslabši rezultat je bil prav tako dosežen na 32-bitnem operacijskem sistemu z uporabo samo ene stalne delovne niti.

Rezultati kažejo, da sta čas, potreben za kreiranje novih nestalni delovnih niti, in čas preverjanja, ali je kakšna delovna nit prosta, velik dejavnik pri zelo hitrih operacijah in zmanjšata zmogljivost in učinkovitost aplikacije.

Za kopiranje manjših datotek je aplikacija najbolj učinkovita, če uporablja 5 stalnih in 5 nestalnih delovnih niti.

Poraba delovnega spomina se je z večanjem števila delovnih niti povečevala, kar je tudi razvidno iz slike 13.

Na sliki 13 so grafično prikazani rezultati testov testnega primera 1 na 64-bitnem sistemu.



Slika 13: Rezultati testnega primera 1

Testni primer 2

V testnem primeru 2 je bilo ugotovljeno, da je kopiranje večjih datotek počasnejša operacija, ki tudi veliko bolj obremeni trdi disk.

V tem testnem primeru so bili boljši rezultati na 64-bitnem operacijskem sistemu doseženi pri uporabi 5 stalnih in 5 oziroma 25 nestalnih delovnih nitih. Na 32-bitnem operacijskem sistemu pa so bili boljši časi doseženi pri uporabi ene same stalne delovne niti in pri uporabi 10 stralnih in 90 nestalnih delovnih niti.

Rezultati posameznih testov so glede na število delovnih niti na obeh operacijskih sistemih tudi v tem testnem primeru primerljivi, saj je vrstni red enak.

Prednost kopiranja manjših datotek na 32-bitnem operacijskem sistemu je tudi v tem testnem primeru manjša poraba delovnega spomina.

Najboljši rezultat kopiranja je bil dosežen na 32-bitnem operacijskem sistemu z uporabo 10 stalnih in 90 nestalnih delovnih niti v nitnem bazenu.

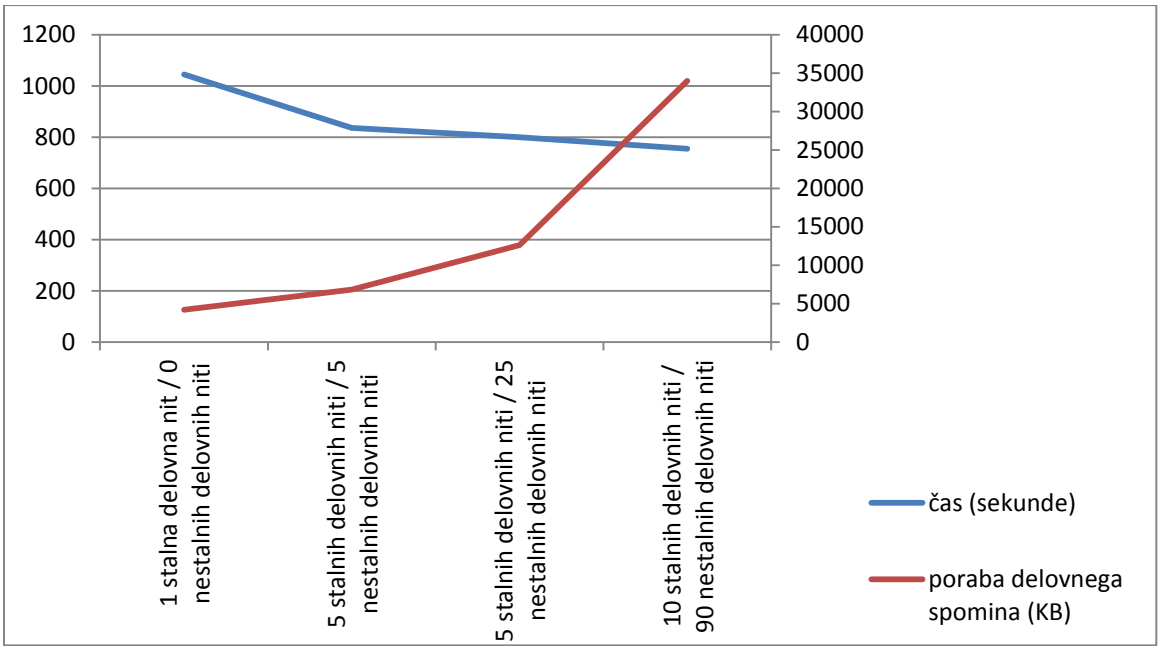
Najslabši rezultat je bil dosežen na 64-bitnem operacijskem sistemu z uporabo samo ene stalne delovne niti.

V tem testnem primeru so se rezultati z večanjem števila delovnih niti v nitnem bazenu izboljševali, kar je lep pokazatelj vpliva večjih nitnih bazenov na učinkovitost aplikacij.

Za kopiranje manjših in večjih datotek je aplikacija najbolj učinkovita, če uporablja večje število stalnih in nestalnih delovnih niti.

Poraba delovnega spomina se je z večanjem števila delovnih niti povečevala, kar je tudi razvidno iz slike 14.

Na sliki 14 so grafično predstavljeni rezultati testov testnega primera 2 na 64-bitnem sistemu.



Slika 14: Rezultati testnega primera 2

Testni primer 3

Kot je bilo že ugotovljeno v testnem primeru 2, je kopiranje večjih datotek počasnejša operacija, ki tudi veliko bolj obremeni trdi disk.

V tem testnem primeru so bili boljši rezultati doseženi na 64-bitnem operacijskem sistemu pri vseh testih, razen v testu z 10 stalnimi in 90 nestalnimi delovnimi nitmi.

Rezultati posameznih testov so glede na število delovnih niti na obeh operacijskih sistemih tudi v tem testnem primeru primerljivi, saj je vrstni red enak.

Prednost kopiranja manjših datotek na 32-bitnem operacijskem sistemu je tudi v tem testnem primeru manjša poraba delovnega spomina.

Najboljši rezultat kopiranja je bil dosežen z uporabo 10 stalnih in 90 nestalnih delovnih niti v nitnem bazenu na 32-bitnem operacijskem sistemu.

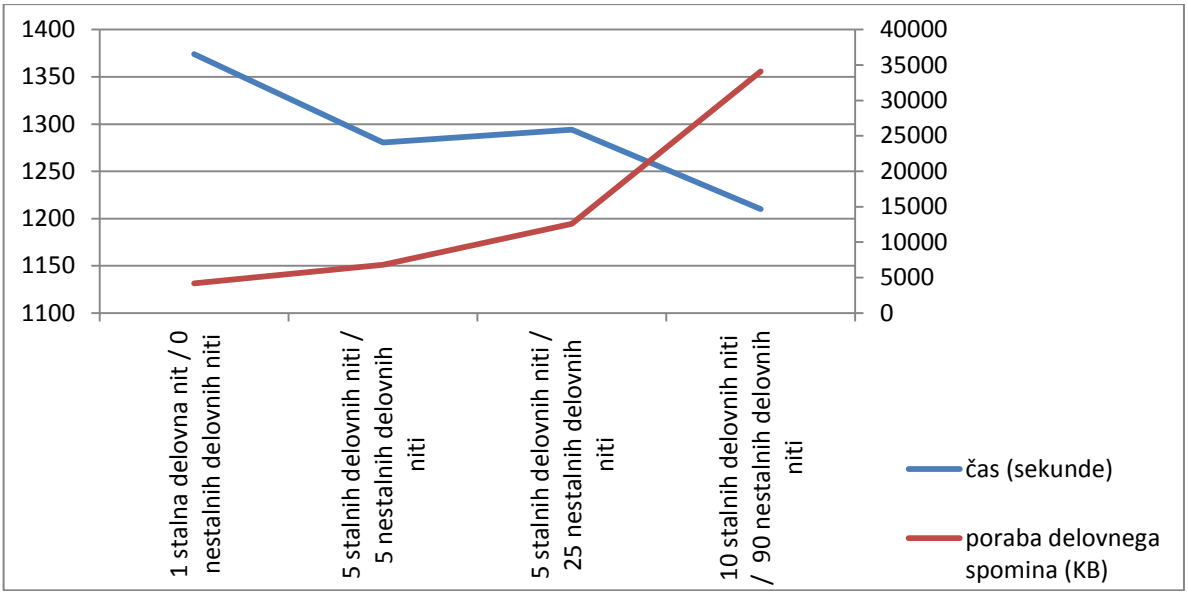
Najslabši rezultat je bil dosežen na 32-bitnem operacijskem sistemu z uporabo samo ene stalne delovne niti.

V tem testnem primeru so se rezultati z večanjem števila delovnih niti v nitnem bazenu sicer izboljševali, vendar manj od pričakovanega.

Za kopiranje večjih datotek je aplikacija najbolj učinkovita, če uporablja 10 stalnih in 90 nestalnih delovnih niti.

Poraba delovnega spomina se je z večanjem števila delovnih niti povečevala, kar je tudi razvidno iz slike 15.

Na sliki 15 so grafično predstavljeni rezultati testov testnega primera 3 na 64-bitnem sistemu.



Slika 15: Rezultati testnega primera 3

Za konec lahko ugotovimo, da je pri kopiranju manjših datotek aplikacija bolj učinkovita z manjšim številom delovnih niti v nitnem bazenu, saj predstavljata čas za kreiranje nove delovne niti in čas za preverjanje, ali je nit prosta, velik dodatek k celotnem času izvajanja kopiranja.

Pri kopiranju večjih in mešanih datotek pa je aplikacija vsekakor učinkovitejša pri uporabi večjega števila delovnih niti v nitnem bazenu.

Sama poraba delovnega spomina je v vseh testih, ki smo jih izvedli, proti pričakovanjem zelo nizka. Največ delovnega spomina je aplikacija uporabljala pri kopiranju večjih datotek z uporabo 10 stalnih in 90 nestalnih delovnih niti, in sicer samo 34 MB, kar je zelo malo.

Proti pričakovanjem pa je zmogljivost trdega diska pri kopiranju večjih datotek predstavljala dejavnik, ki je zmanjševal učinkovitost naše aplikacije, in pokazal, da veliko število delovnih niti v nitnem bazenu ne pomeni učinkovitejše aplikacije.

6 Zaključek

V diplomski nalogi smo razvili enostavno aplikacijo za kopiranje datotek, s katero smo želeli pokazati njeno učinkovitost z uporabo različnega števila delovnih niti v nitnem bazenu. Z aplikacijo smo izvedli različne teste in rezultate tudi primerjali.

Pri načrtovanju in razvoju aplikacije smo pridobili veliko teoretičnega znanja o večnitnih aplikacijah, načinih razvoja večnitnih aplikacij in omejitvah pri razvoju večnitnih aplikacij. Pri samem razvoju nitnega bazena pa smo pridobili še praktično znanje o razvoju večnitnih aplikacij, zaščiti podatkov z uporabo sinhronizacijskih objektov in načinu samega dodeljevanja nalog delovnim nitim.

S samim testiranjem uporabe različnih nitnih bazenov za kopiranje datotek smo potrdili pričakovanja, da je aplikacija učinkovitejša, če uporablja več delovnih niti pri počasnejših operacijah, kot je v našem primeru kopiranje večjih datotek. Kot smo pričakovali, se uporaba večjega števila niti pri hitrejših operacijah ni izkazala za učinkovitejšo rešitev. Je pa v primerjavi z uporabo samo ene stalne delovne niti aplikacija učinkovitejša, če uporablja več delovnih niti. Malce nepričakovane rezultate so pokazali testi kopiranja večjih datotek. Večje število delovnih niti je sicer prineslo malenkost večjo učinkovitost, vendar je sam trdi disk oziroma njegova zmogljivost in odzivnost predstavljal veliko omejitev pri učinkovitosti kopiranja aplikacije.

Samo učinkovitost aplikacije bi lahko izboljšali z malo drugačnim načinom kopiranja datotek. Trenutna verzija aplikacije najprej prebere celotno vsebino imenika in si zapomni imena datotek v imeniku, potem pa za vsako datoteko izvede kopiranje (branje in pisanje). Velika verjetnost je, da bi bila aplikacija učinkovitejša, če bi določen del delovnih niti bral vsebino datotek imenika, drugi del delovnih niti pa sočasno že pisal prebrane datoteke.

Glede na to, da so se v vseh testih ustvarile vse možne nestalne delovne niti, bi bilo zanimivo izvesti tudi teste z večjim številom stalnih in manjšim številom nestalnih delovnih niti v nitnem bazenu.

Sama aplikacija je zasnovana tako, da bi lahko z majhnimi spremembami v kodi namesto kopiranja datotek izvajali kakšno drugo nalogo, ki je na primer manj odvisna od zmogljivosti trdega diska. Rezultati testov bi lahko bili povsem drugačni.

Izkušnje, ki smo jih v tem delu pridobili, nam bodo pomagale predvsem v praksi, saj večina novejših aplikacij za svoje delovanje uporablja več niti ali celo več procesov.

Literatura

[1] D. R. Butenhof: Programming with POSIX Threads, Združene države Amerike, Addison-Wesley, 1997, pogl. 1 in 2.

[2] A. Williams: C++ Concurrency in action (Practical multithreading), Združene države Amerike, Manning publication Co., 2009, pogl. 1-4.

[3] B. Lewis, D. J. Berg, PThreads Primer, A Guide to Multithreaded Programming, Združene države Amerike, SunSoft Press, 1996, pogl. 6.

[4] (2011) J. Niu, Concurrency: Mutual Exclusion and Synchronization - Part 1, October 8, 2003. Dostopno na: http://web.cs.gc.cuny.edu/~jniu/teaching

[5] (2011) D. Xu, Performance study and dynamic optimization design for thread pool systems, 2004. Dostopno na: www.osti.gov/bridge/servlets/purl/835380-ZOcXfL/webviewable/

[6] (2011) T. Richardson, The OMNI Thread Abstraction, AT&T Laboratories Cambridge, November 2001. Dostopno na: http://omniorb.sourceforge.net/omni40/omnithread.html

[7] (2011) The Computer Science Department, Operating Systems, Deadlock, 2004. Dostopno na: http://www.cs.rpi.edu/academics/courses/fall04/os/c10/

[8] (2011) M. Russinovich, Pushing the Limits of Windows: Processes and Threads, 2009. Dostopno na: http://blogs.technet.com/b/markrussinovich/archive/2009/07/08/3261309.aspx

[9] (2011) R. Barry, RTOS Task Switching: An Example Implementation In C, 2004. Dostopno na: http://www.eetimes.com/electronics-news/4196932/RTOS-Task-Switching-An-Example-Implementation-In-C

[10] (2011) Thread pool pattern. Dostopno na: http://en.wikipedia.org/wiki/Thread_pool_pattern

[11] (2011) H. L. Singh, Running Multiple Threads in Cocoa, 2001. Dostopno na: http://cocoadevcentral.com/articles/000061.php

[12] (2011) C. Schock Multithreading. Dostopno na:

http://pages.cpsc.ucalgary.ca/~schock/wiki/index.php/CPSC219_Multithreading

[13] (2011) C++ MultiThreading tuturial.
Dostopno na: http://www.paulbridger.com/mutexes/

[14] (2011) Virtual address space. Wikipedia.
Dostopno na: http://en.wikipedia.org/wiki/Virtual_address_space

[15] (2011) OMNIThreade abstrakcija. Dostopno na: http://omniorb.sourceforge.net/