**SVEUČILIŠTE U SPLITU**

**FAKULTET ELEKTROTEHNIKE, STROJARSTVA I BRODOGRADNJE**

NAPREDNE ARHITEKTURE RAČUNALA (250)

IZVJEŠTAJ 1

**Toni Biuk**

**Tino Melvan**

**Split, travanj 2018.**

**SADRŽAJ**

[1. Specifikacije računala 1](#_Toc511080094)

[2. Zadatak 1 – Pthread niti 2](#_Toc511080095)

[2.1. Sekvencijalno rješenje 2](#_Toc511080096)

[2.2. Paralelno rješenje 2](#_Toc511080097)

[2.3. Rezultati 4](#_Toc511080098)

[3. Zadatak 2 - Mutex 7](#_Toc511080099)

[3.1. Rješenje 7](#_Toc511080100)

[3.2. Problemi 8](#_Toc511080101)

[4. Prilog 9](#_Toc511080102)

[4.1. Spora funkcija računanja sume 9](#_Toc511080103)

[4.2. Traces.hpp 9](#_Toc511080104)

[4.3. Random.hpp 10](#_Toc511080105)

[4.4. Random.cpp 10](#_Toc511080106)

# Specifikacije računala

U ovom poglavlju navedene su relevantne specifikacije korištenih računala na kojima su se izvodili programi navedeni u ostalim poglavljima ili dani u nastavku. Specifikacije se nalaze u tablicama ispod.

Tablica 1. Specifikacije računala 1.

|  |  |
| --- | --- |
| Operacijski sustav | Microsoft Windows 10 Education Build 16299.309 |
| Procesor | Intel® Core™ i7-7700HQ @ 2.80GHz |
| Broj jezgri procesora | 4 |
| Broj niti procesora | 8 |
| RAM | 16GB (2X8GB) DDR4 – 2400MHz |

Tablica 2. Specifikacije računala 2.

|  |  |
| --- | --- |
| Operacijski sustav | Microsoft Windows 10 Pro Build 16299 |
| Procesor | AMD FX-6300 @ 3.50GHz |
| Broj jezgri procesora | 3 |
| Broj niti procesora | 6 |
| RAM | 16 GB (8x2) DDR3 – 1066Mhz |

# Zadatak 1 – Pthread niti

## Sekvencijalno rješenje

#include <Utility/Random.hpp>

#include <Utility/Traces.hpp>

#include <chrono>

#include <cstdint>

#include <cstdlib>

#define ARRAY\_LENGTH 100000

#define ARRAY\_MIN\_RANGE (-1000)

#define ARRAY\_MAX\_RANGE 1000

typedef std::chrono::high\_resolution\_clock Clock;

int32\_t main()

{

const Random random;

int32\_t array[ARRAY\_LENGTH];

auto result = 0;

for (auto& num : array)

{

num = random.Next(ARRAY\_MIN\_RANGE, ARRAY\_MAX\_RANGE);

DEBUG("[Main] Adding to array: %d\n", num);

}

const auto clockStart = Clock::now();

for (const auto& num : array)

{

result += num;

}

const auto clockEnd = Clock::now();

INFO("[Main] Execution time: %lld microsec\n",

std::chrono::duration\_cast<std::chrono::microseconds>

(clockEnd - clockStart).count());

INFO("[Main] Final result: %d\n", result);

return EXIT\_SUCCESS;

}

## Paralelno rješenje

#include <Utility/Random.hpp>

#include <Utility/Traces.hpp>

#define HAVE\_STRUCT\_TIMESPEC

#include <pthread.h>

#include <chrono>

#include <cstdint>

#include <cstdlib>

#include <cmath>

#define ARRAY\_LENGTH 100000

#define ARRAY\_MIN\_RANGE (-1000)

#define ARRAY\_MAX\_RANGE 1000

#define NUMBER\_OF\_THREADS 4

struct ArgStruct {

int32\_t threadId;

int32\_t \*arrayStart;

int32\_t \*arrayEnd;

int32\_t result;

};

typedef std::chrono::high\_resolution\_clock Clock;

void\* GetPartialSum(void \*arguments)

{

auto \*args = static\_cast<struct ArgStruct \*>(arguments);

auto result = 0;

DEBUG("[Thread %d] Started...\n", args->threadId);

DEBUG("[Thread %d] Starting with %d\n", args->threadId, \*args->arrayStart);

DEBUG("[Thread %d] Ending with %d\n", args->threadId, \*args->arrayEnd);

while (args->arrayStart <= args->arrayEnd)

{

result += \*args->arrayStart;

args->arrayStart++;

}

args->result = result;

DEBUG("[Thread %d] Calculated %d\n", args->threadId, result);

return nullptr;

}

int32\_t main()

{

DEBUG("[Main] Starting program...\n");

DEBUG("[Main] ARRAY\_LENGTH: %d\n", ARRAY\_LENGTH);

DEBUG("[Main] NUMBER\_OF\_THREADS: %d\n", NUMBER\_OF\_THREADS);

const Random random;

struct ArgStruct args[NUMBER\_OF\_THREADS];

pthread\_t threads[NUMBER\_OF\_THREADS];

int32\_t array[ARRAY\_LENGTH];

auto result = 0;

const auto range = static\_cast<int>

(static\_cast<float>(ARRAY\_LENGTH) / NUMBER\_OF\_THREADS);

DEBUG("[Main] Division range: %d\n", range);

DEBUG("[Main] Making random array...\n");

for (auto& num : array)

{

num = random.Next(ARRAY\_MIN\_RANGE, ARRAY\_MAX\_RANGE);

DEBUG("[Main] Adding to array: %d\n", num);

}

const auto clockStart = Clock::now();

for (auto id = 0; id < NUMBER\_OF\_THREADS; id++)

{

args[id].threadId = id;

args[id].arrayStart = array + id \* range;

(id == NUMBER\_OF\_THREADS - 1) ? args[id].arrayEnd = &array[ARRAY\_LENGTH - 1]

: args[id].arrayEnd = array +

(id + 1) \* range - 1;

const auto rc = pthread\_create(&threads[id],

nullptr,

GetPartialSum,

&args[id]);

if (rc)

{

INFO("[Main] ERROR; return code from pthread\_create() is %d\n", rc);

return EXIT\_FAILURE;

}

}

DEBUG("[Main] Joining threads!\n");

for (const auto& t : threads)

{

pthread\_join(t, nullptr);

}

DEBUG("[Main] Calculating result!\n");

for (const auto& arg : args)

{

result += arg.result;

}

const auto clockEnd = Clock::now();

INFO("[Main] Execution time: %lld microsec\n",

std::chrono::duration\_cast<std::chrono::microseconds>

(clockEnd - clockStart).count());

INFO("[Main] Final result: %d\n", result);

return EXIT\_SUCCESS;

}

## Rezultati

Zadatak je, dakle, izračunati sumu niza, i to prvo sekvencijalno (poglavlje 2.1), a potom paralelizirati problem i iskoristiti niti (poglavlje 2.2). U oba koda, veličina niza definirana je s *ARRAY\_LENGTH*, a broj niti kod paralelizacije s *NUMBER\_OF\_THREADS*. Sama paralelizacija izvedena je tako da se niz dijeli na dijelove, ovisno o željenom broju niti, a zatim se pojedinačne sume spoje u konačni rezultat.

U nastavku dan je graf koji prikazuje prosječno izvršenje programa danih u poglavljima 2.1 te 2.2. Prema njemu, očigledna je činjenica da je sekvencijalno izvođenje danog programa višestruko brže od paralelnog, te da se samim povećanjem broja niti ujedno i vrijeme izvršavanja paralelnog programa samo povećanje. Razlog je vrlo jednostavan. Ako pokrećemo mali i relativno jednostavni program, samo pokretanje niti, organiziranje prioriteta izvršavanja te izmjena konteksta traju višestruko više nego sami proces koji se želi obaviti. U ovom konkretnom slučaju, vrijeme izračuna sume zanemarivo je u usporedbi s navedenim. Zbog toga niti ne obavljaju dovoljnu količinu posla te nisu isplative.

Slika 2.1. Brzina izvršavanja

Kako bi niti postale isplative, one moraju obavljati više posla. Upravo zbog toga program je izmijenjen tako da se funkcija računanja sume usporila. Novi kod može se vidjeti u prilogu (poglavlje 4.1). Novo vrijeme izvođenja prikazano je na slici 2.2. U novonastaloj funkciji vrijeme izvođenja više nije trivijalno te se ono linearno smanjuje porastom broja niti, ako je broj niti manji od osam. Magični broj osam je tu zbog toga što procesor korištenog računala podržava istovremeno izvođenje osam niti. Dodatnim povećanjem poslije tih osam niti brzina izvođenja ostaje ista, ili se relativno malo smanjuje.

Slika 2.2. Spora funkcija, računalo 1.

Na slici 2.3 prikazano je izvođenje iste funkcije na drugome računalu. Za razliku od prvog primjera, ovo računalo ima trojezgreni procesor koji istovremeno podržava izvođenje šest niti. Rezultati su slični, samo se u ovom slučaju ubrzanje prestaje primjećivati nakon šest niti, što dodatno potkrepljuje prvotnu hipotezu.

Slika 2.3. Spora funkcija, računalo 2.

# Zadatak 2 - Mutex

## Rješenje

#include <Utility/Traces.hpp>

#include <cstdint>

#include <mutex>

#include <thread>

#include <vector>

#define NUM\_OF\_ITERATIONS 1000U

#define NUM\_OF\_THREADS 8U

void Increment(uint32\_t &n, std::mutex &mutex)

{

for (auto i = 0U; i < NUM\_OF\_ITERATIONS; i++)

{

mutex.lock();

n++;

DEBUG("[Random thread] Current n: %u\n", n);

mutex.unlock();

}

}

int32\_t main()

{

auto val = 0U;

std::vector<std::thread> threads;

std::mutex mutex;

for (auto i = 0U; i < NUM\_OF\_THREADS; i++)

{

threads.emplace\_back(std::thread(Increment,

std::ref(val),

std::ref(mutex)));

}

for (auto& thread : threads)

{

thread.join();

}

INFO("%u\n", val);

return EXIT\_SUCCESS;

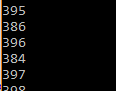
}

## Problemi

Ako u kodu iz rješenja izuzmemo zaštitu uz pomoć mutexa, može doći do velikih problema prilikom izvođenja niti. Prvi problem na koji se može naići jest stanje nadmetanja (engl. *race condition).* To je stanje u kojem dvije ili više niti pokušavaju istovremeno izvršiti povezani blok koda, uzrokujući pri tome različiti rezultat izvođenja programa ovisno o redoslijedu samog izvršavanja. U ovom primjeru to bi se očitovalo različitim redoslijedom ispisa trenutne vrijednosti varijable.

Drugi problem koji se može pojaviti, iako danas rijetko zbog brzine izvođenja operacija od strane procesora, jest trka za podacima (engl. *data race*). Ono se događa kada dvije ili više niti pokušavaju istovremeno pristupiti istoj memorijskoj lokaciji i promijeniti njenu vrijednost. U primjeru to bi bio pokušaj istovremenog inkrementiranja od strane više niti. Posljedica je gubljenje točnosti (ili u potpunosti krivi) rezultata.

Ovakvi problemi mogu se riješiti na više načina, a u primjeru je odabran mutex koji predstavlja isključujuću zastavicu. Ona radi tako da propušta samo jednu nit u kritični dio koda, dok blokira sve druge sve dok se mutex nije otključao. Tada se iduća nit propušta i mutex ponovo zaključava. Na ovaj način imamo osiguranje da će kritični kod biti obrađivan samo od strane jedne niti.



Slika 3.1. Primjer krivog ispisa

# Prilog

## Spora funkcija računanja sume

void\* GetPartialSum(void \*arguments)

{

auto \*args = static\_cast<struct ArgStruct \*>(arguments);

auto result = 0;

DEBUG("[Thread %d] Started...\n", args->threadId);

DEBUG("[Thread %d] Starting with %d\n", args->threadId, \*args->arrayStart);

DEBUG("[Thread %d] Ending with %d\n", args->threadId, \*args->arrayEnd);

while (args->arrayStart <= args->arrayEnd)

{

result += \*args->arrayStart;

args->arrayStart++;

// simulate extra workload

for (auto i = 0U; i < 300U; i++)

{

time(nullptr);

}

}

args->result = result;

DEBUG("[Thread %d] Calculated %d\n", args->threadId, result);

return nullptr;

}

## Traces.hpp

#ifndef UTILITY\_TRACES\_HPP

#define UTILITY\_TRACES\_HPP

#include <cstdio>

#define INFO(fmt, ...) do { fprintf(stderr, fmt, \_\_VA\_ARGS\_\_); } while (0);

#if \_DEBUG

#define DEBUG(fmt, ...) do { INFO(fmt, \_\_VA\_ARGS\_\_); } while (0);

#else

#define DEBUG(mt, ...)

#endif

#endif // !UTILITY\_TRACES\_HPP

## Random.hpp

#ifndef UTILITY\_RANDOM\_HPP

#define UTILITY\_RANDOM\_HPP

#include <cstdint>

#include <random>

class Random

{

public:

Random();

explicit Random(uint32\_t seed);

int32\_t Next() const;

int32\_t Next(int32\_t ceiling) const;

int32\_t Next(int32\_t floor, int32\_t ceiling) const;

double NextDouble() const;

float NextFloat() const;

private:

std::default\_random\_engine& globalURNG() const;

void Randomize() const;

void Randomize(uint32\_t seed) const;

};

#endif // !UTILITY\_RANDOM\_HPP

## Random.cpp

#include "Utility/Random.hpp"

Random::Random()

{

Randomize();

}

Random::Random(const uint32\_t seed)

{

Randomize(seed);

}

int32\_t Random::Next() const

{

return Next(0, INT32\_MAX);

}

int32\_t Random::Next(const int32\_t ceiling) const

{

return Next(0, ceiling);

}

int32\_t Random::Next(const int32\_t floor, const int32\_t ceiling) const

{

static std::uniform\_int\_distribution<> d{};

using parm\_t = decltype(d)::param\_type;

if (floor < ceiling)

{

return d(globalURNG(), parm\_t{ floor, ceiling });

}

return d(globalURNG(), parm\_t{ ceiling, floor });

}

double Random::NextDouble() const

{

return static\_cast<double>(Next()) / INT32\_MAX;

}

float Random::NextFloat() const

{

return static\_cast<float>(Next()) / INT32\_MAX;

}

std::default\_random\_engine& Random::globalURNG() const

{

static std::default\_random\_engine u{};

return u;

}

void Random::Randomize() const

{

static std::random\_device rd{};

globalURNG().seed(rd());

}

void Random::Randomize(const uint32\_t seed) const

{

globalURNG().seed(seed);

}