

**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
2. Zadatak 1 – Pthread niti .....	2
2.1. Sekvencijalno rješenje .....	2
2.2. Paralelno rješenje.....	2
2.3. Rezultati.....	4
3. Zadatak 2 - Mutex .....	7
3.1. Rješenje .....	7
3.2. Problemi.....	8
4. Prilog.....	9
4.1. Spora funkcija računanja sume.....	9
4.2. Traces.hpp.....	9
4.3. Random.hpp.....	10
4.4. Random.cpp.....	10

## 1. 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.*

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

*Tablica 2. Specifikacije računala 2.*

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

## 2. Zadatak 1 – Pthread niti

### 2.1. 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;
}
```

### 2.2. 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;
}

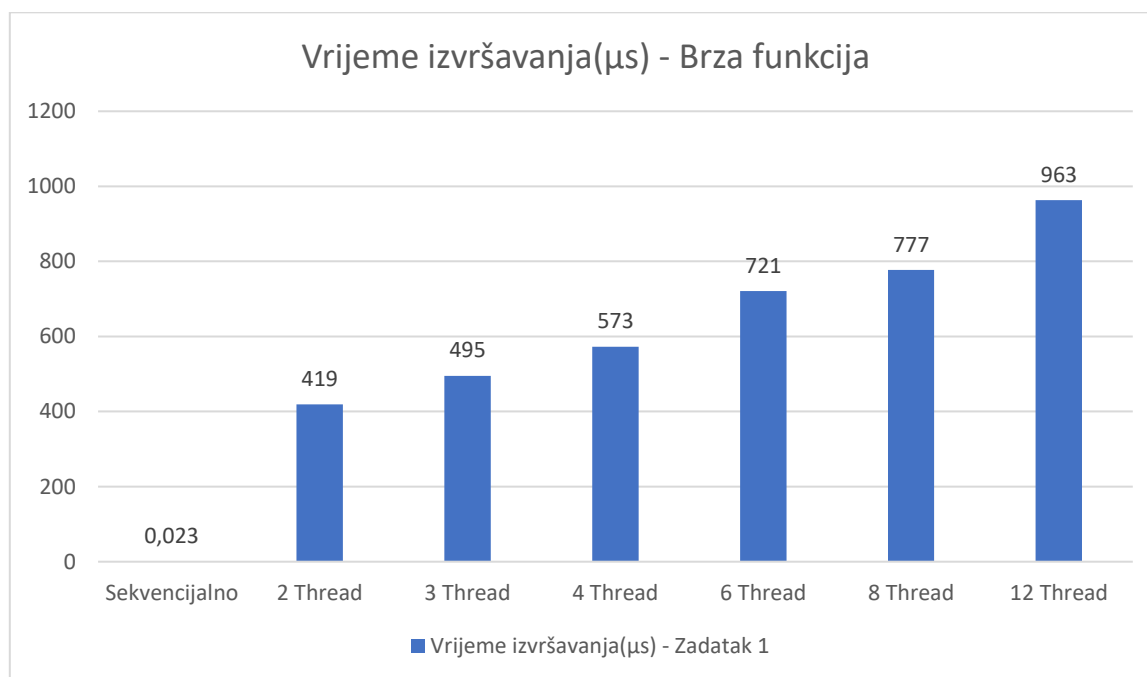
```

## 2.3. Rezultati

Zadatak je, dakle, izračunati sumu niza, i to prvo sekvencijalno (poglavljje 2.1), a potom paralelizirati problem i iskoristiti niti (poglavljje 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.

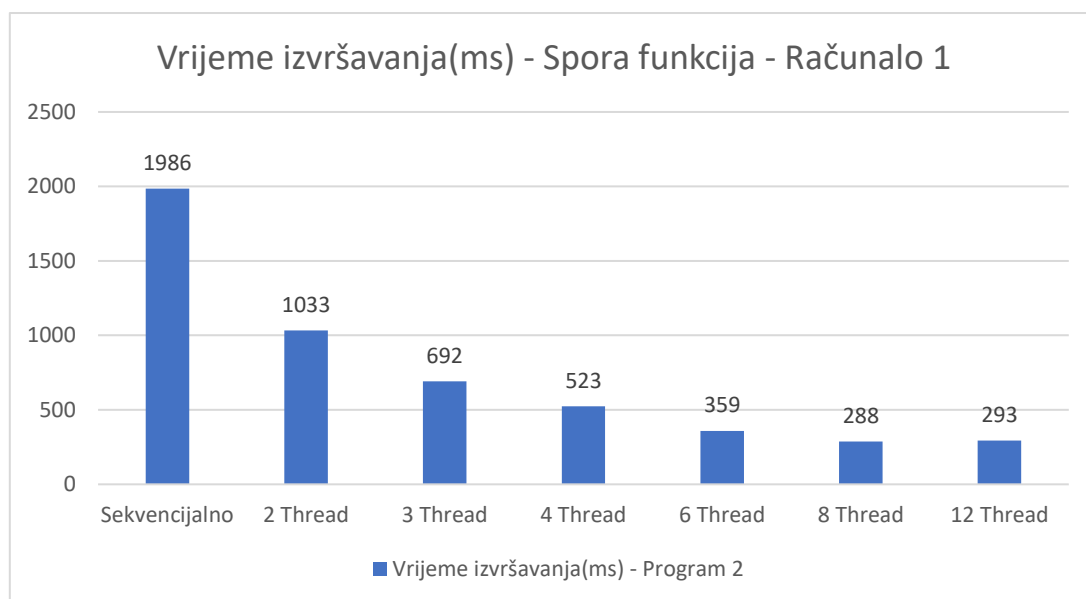
U nastavku dan je graf koji prikazuje prosječno izvršenje programa danih u poglavljima 2.1 te 2.2. Prema njemu, očigledna je činjenica da je sekvencijalno izvođenje danog programa višestruko brže od paralelnog, te da se samim povećanjem broja niti ujedno i vrijeme izvršavanja paralelnog programa samo 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.



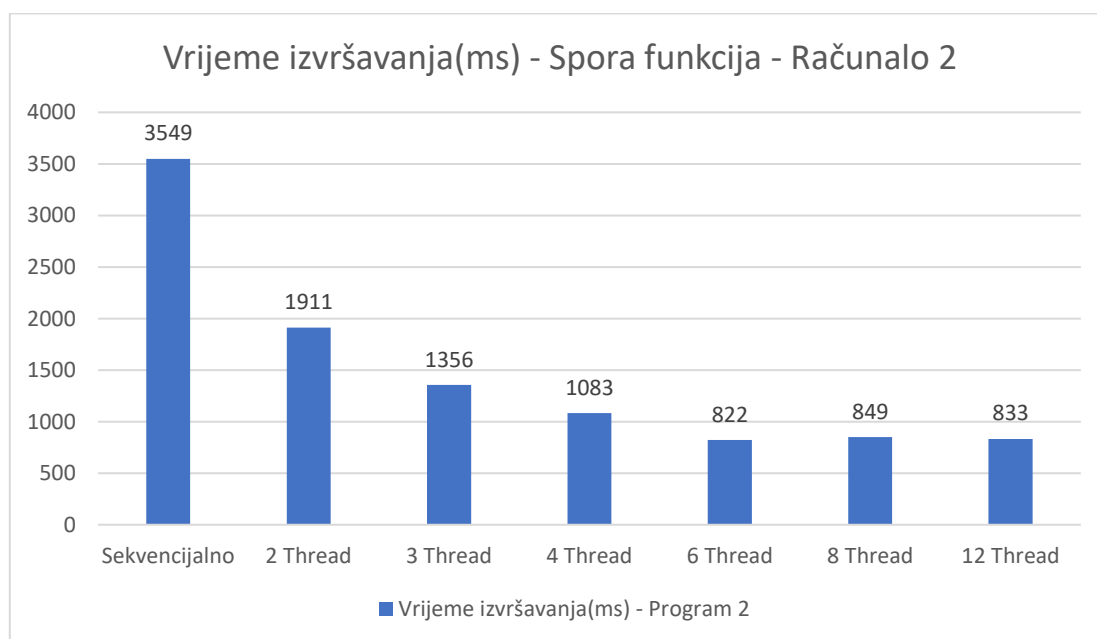
*Slika 2.1. Brzina izvršavanja*

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.*



### 3. Zadatak 2 - Mutex

#### 3.1. 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);

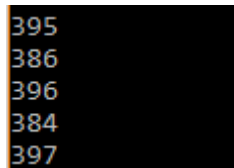
    return EXIT_SUCCESS;
}
```

### 3.2. 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.



```
395
386
396
384
397
```

*Slika 3.1. Primjer krivog ispisa*

## 4. Prilog

### 4.1. 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;
}
```

### 4.2. 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
```

### 4.3. Random.hpp

```
#ifndef UTILITY_RANDOM_HPP
#define UTILITY_RANDOM_HPP

#include <stdint>
#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
```

### 4.4. 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);
}

```