Masurarea timpului de executie a proceselor in diferite limbaje de programare (C, C++, Java)

Giesswein Alexia

Grupa: 30231

Octombrie 2022

Cuprins

1. Introducere

1.1 Context

Scopul acestui proiect este de a masura timpul de executie a proceselor in diferite limbaje de programare. Programele vor fi scrise in cele trei limbaje de programare alese, C, C++ si Java, si se va face o comparatie intre masuratorile facute pentru acestea. Se va masura timpul de executie pentru alocarea de memorie si accesul la memorie (atat static, cat si dinamic), pentru crearea unui thread, pentru thread context switch si pentru thread migration in toate cele trei limbaje de programare, iar aceste masuratori vor fi comparate.

Aceste masuratori ale timpului de executie ale diferitelor procese si compararea celor trei limbaje de programare pot fi utile atunci cand este nevoie de o anumita eficienta, iar testand si calculand timpul de executie se poate decide care limbaj de programare este mai util pentru un anumit scop.

1.2 Specificatii

Programele vor fi scrise in Intellij IDEA, pentru limbajul de programare Java si in Visual Studio, pentru limbajele C, C++. Acestea vor fi capabile sa masoare si sa afiseze timpul de executie pentru alocarea de memorie, accesul la memorie, crearea unui thread, thread context switch si thread migration, care ulterior vor fi comparate pentru a vedea diferentele dintre limbajele de programare.

1.3 Obiective

Obiectivele acestui proiect sunt proiectarea si implementarea unor programe care sa masoare timpul de executie in trei limbaje de programare diferite, iar rezultatele finale vor fi interpretate si comparate. Trebuie implementate niste functii care aloca memorie si acceseaza memoria (care se implementeaza diferit in functie de limbaj), functii pentru crearea unui thread, thread migration (care muta un thread dintr-un core in altul) si thread context switch (care schimba intre ele doua threaduri).

2. Studiu bibliografic

2.1 Masurarea timpului de executie al unui program

Masurarea timpului de executie a unui program runtime sau compilat este mai dificila decat se presupune, deoarece multe metode nu sunt de multe ori transferabile pe alte platforme. Alegerea metodei potrivite va depinde de sistemul de operare si de limbajul de programare. [1]

Specificatiile laptopului de pe care voi face masuratorile: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz, 64-bit operating system, x64-based processor, Windows 10 Home.

Este important sa definim si să facem diferența între cei doi termeni, Wall Time și CPU Time, care sunt adesea folosiți atunci când se măsoară timpul de rulare. Wall Time (”digital clock”) este pur si simplu timpul total scurs in timpul unei masuratori, care poate fi masurat cu un cronometru, presupunand ca se poate porni si opri exact in punctele de executie dorite. CPU Time se refera la timpul in care un CPU procesa instructiunile programelor. Timpul petrecut în asteptarea altor lucruri (de exemplu, operații I/O) nu este inclus în timpul CPU.

Astfel, Wall Time masoara cat timp a trecut, iar CPU Time cate secunde a fost ocupat un CPU. [1]

Pentru a măsura timpul de execuție, se poate efectua o masurare în două momente de timp diferite pe ciclurile de ceas trecute de la un anumit început. Atunci, prin găsirea diferenței și cunoașterea CLOCKS_PER_SEC, se pot calcula secundele trecute de la primul apel al functiei clock(), la al doilea apel. Acesta este rezultatul functiei clock().[1]

```
clock_t start = clock();

// let's do some operations

clock_t end = clock();

long double seconds = (float)(end - start) / CLOCKS_PER_SEC;
```

Figura 1 Masurarea timpului de executie cu clock(); [1]

Timpul CPU este reprezentat de tipul de date clock_t, care este numărul de cicluri ale semnalului de ceas. Prin împărțirea numărului de cicluri de ceas la ceasuri pe secundă rezulta cantitatea totală de timp în care un proces a folosit în mod activ un procesor după un eveniment arbitrar. CLOCKS_PER_SEC este deja definit de compilator și este necesar să fie inclusa biblioteca time.h. [1]

2.2 Alocarea de memorie si accesul la memorie

Alocarea memoriei este procesul de a pune deoparte secțiuni de memorie într-un program pentru a fi utilizate pentru a stoca variabile și instanțe de structuri și clase. Există două tipuri de bază de alocare a memoriei: alocare statica si dinamica. [2]

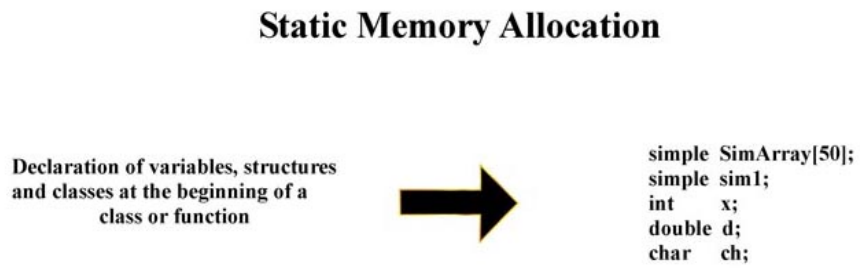


Figura 2 Alocare statica a memoriei [2]

La alocare statica, memoria pentru obiectul declarat este alocată de sistemul de operare. [2] Numele declarat al obiectului poate fi apoi folosit pentru a accesa acel bloc de memorie. [2]

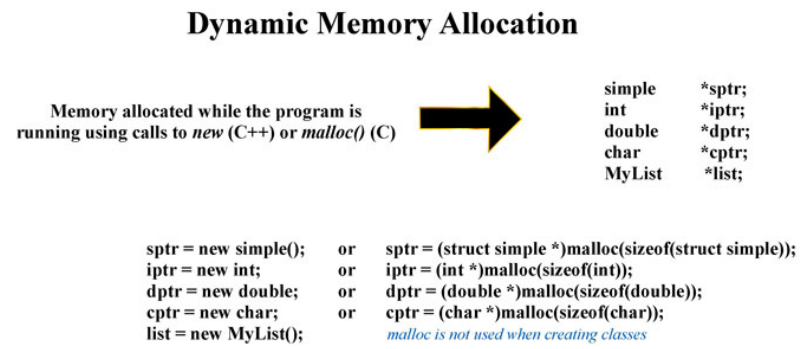


Figura 3 Alocare dinamica a memoriei [2]

La alocarea dinamică a memoriei, sistemul de operare trebuie să desemneze un bloc de memorie de dimensiunea corespunzătoare în timp ce programul rulează. Acest lucru se face fie cu noul operator, fie cu un apel la funcția malloc. Blocul de memorie este alocat și este returnat un pointer către bloc. Acesta este apoi stocat într-un pointer de tipul de date corespunzător. [2]

Accesul la memorie se poate face prin mai multe metode, precum: acces secvential (într-o manieră secvenţială liniară specifică, cum ar fi accesarea într-o lista simplu inlantuita. Timpul de acces depinde de locația datelor), acces random (in această metodă, orice locație a memoriei poate fi accesată aleatoriu, precum accesarea într-un vector. Locațiile fizice sunt independente în această metodă de acces), acces direct (in această metodă, blocurile sau înregistrările individuale au o adresă unică bazată pe locația fizică. [3] Accesul direct la

memorie (DMA) este o metodă care permite unui dispozitiv de intrare/ieșire (I/O) să trimită sau să primească date direct către sau din memoria principală, ocolind CPU pentru a accelera operațiunile de memorie. Timpul de acces depinde atât de organizarea memoriei, cât și de caracteristicile tehnologiei de stocare. Accesul este semi-aleatoriu sau direct) [4] si acces asociat (in aceasta metoda, un cuvânt este accesat mai degrabă decât adresa sa. Este un tip special de metodă de acces aleatoriu. Aplicația de acces la memorie asociată este memoria cache). [3]

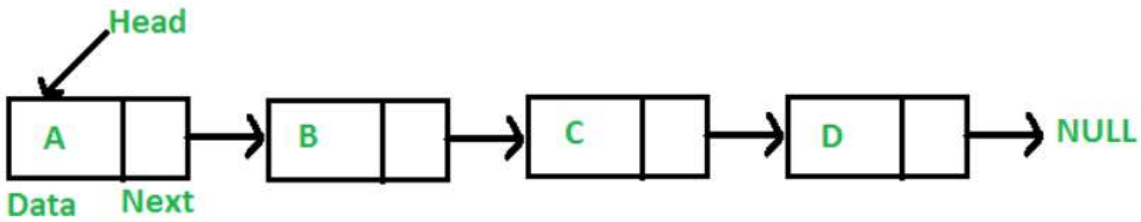


Figura 4 Acces secvential (lista simplu inlantuita) [3]

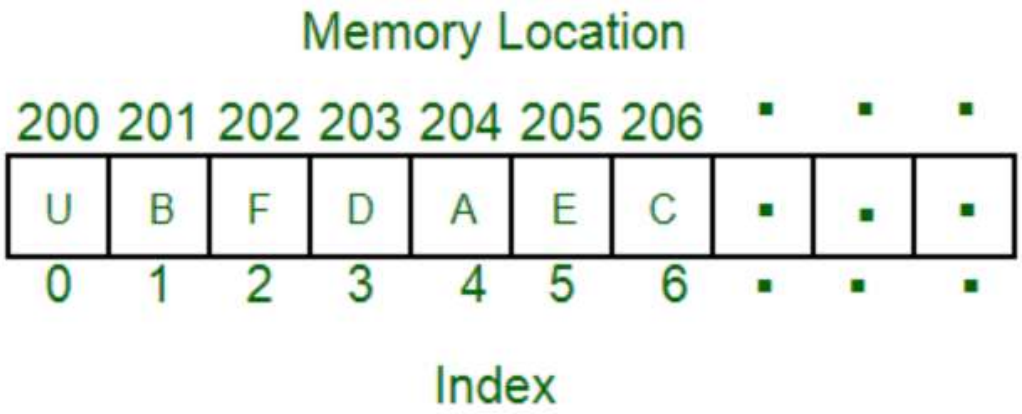


Figura 5 Acces random [3]

Se va face masurarea timpului de executie a alocarii de memorie si accesului la memorie in limbajele de programare C, C++ si Java.

2.3 Crearea unui thread, thread context switch, thread migration

Firul de execuție (thread) este cea mai mică secvență de instrucțiuni programate care poate fi gestionată independent de un planificator, care este de obicei o parte a sistemului de operare. Implementarea threadurilor și a proceselor diferă între sistemele de operare, dar în majoritatea cazurilor un thread este o componentă a unui proces. [5] Un thread se creeaza cu functia pthread_create(), care are 4 parametri (pointer la ID-ul threadului, atributele threadului, numele functiei pe care o va executa threadul si argumentele de inceput). [6]
Context switching este procesul de stocare a stării unui thread, astfel încât acesta să poată fi restaurat pentru a relua execuția la un moment ulterior în timp. Context switching rapid între threaduri este costisitoare în ceea ce privește utilizarea CPU. Pentru fiecare context switch, kernel durează aproximativ 5 µs (în medie) pentru procesare. Thread switch este un tip de context switch care trece de la un thread la altul în același proces. Thread switching este foarte eficientă, deoarece implică comutarea numai a identităților și a resurselor, cum ar fi program counter, registrele și stack pointers. [7]
Thread migration în terminologia sistemulor de operare înseamnă de fapt mutarea threadului dintr-o coadă de rulare a unui core la alta. Acest lucru este realizat de programatorul sistemului de operare (sau planificatorul libpthread în cazul threadurilor PTHREAD_SCOPE_PROCESS) pentru a echilibra încărcarea cozilor de rulare ale diferitelor nuclee (core) disponibile. [8]
Se va face masurarea timpului de executie a crearii unui thread, thread context switch si thread migration in limbajele de programare C, C++ si Java.

3. Analiza

3.1 Propunere proiect

Propunerea mea pentru acest proiect este de a crea o aplicatie care contine mai multe programe care masoara timpul de executie a diferitelor procese, precum alocarea de memorie, accesul la memorie, crearea unui thread, thread context switch si thread migration, in diferite limbaje de programare (C, C++, Java) si compara acesti timpi de executie. Exista multe diferente intre limbajele de programare, procesele descrise mai sus (alocarea de memorie, accesul la memorie, crearea threadurilor, thread context switch si thread migration) se fac diferit in functie de limbajul de programare ales, iar prin aceste masuratori a timpului de executie vom face o comparatie intre limbajele de programare si vom observa care procese sunt mai rapide in functie de limbajul de programare (C, C++, Java) si care limbaj de programare este mai util pentru o problema data. Aplicatia va avea o interfata grafica usor de utilizat, care va afisa timpii de executie a proceselor pentru fiecare limbaj de programare si va afisa care limbaj este mai eficient pentru un anumit proces (are timpul de executie mai mic).

3.2 Plan de proiect

Pentru realizarea acestui proiect, am conceput urmatoarea planificare:

- Saptamana 2, 10.10.2022 : Alegerea proiectului
- Saptamana 3: Scrierea introducerii si a studiului bibliografic in documentatie, intelegerea notiunilor si a modului de masurare a timpilor de executie
- Saptamana 4, 24.10.2022 : Prezentarea documentiei, introducerea si studiul bibliografic

- Saptamana 5: Analiza cerintelor, propunerea proiectului (ce functionalitati va avea)
- Saptamana 6, 7.11.2022 : Prezentarea documentatiei, analiza cerintelor
- Saptamana 7: Proiectarea aplicatiei, cum va arata aplicatia, structura de baza
- Saptamana 8, 21.11.2022 : Prezentarea documentatiei, proiectarea
- Saptamana 9: Implementarea programelor care masoara timpii de executie a proceselor in cele 3 limbaje de programare alese si a interfetei grafice
- Saptamana 10, 5.12.2022 : Prezentarea documentatiei si implementarea
- Saptamana 11: Testarea si validarea aplicatiei, facand cat mai multe teste pentru a verifica functionalitatea acesteia
- Saptamana 12, 19.12.2022 : Prezentarea documentatiei, testarea si validarea
- Saptamana 13 si saptamana 14: Finalizarea aplicatiei, concluzii si bibliografia finala
- Saptamana 15: Prezentare proiect, documentatia si aplicatia finala

3.3 Analiza/Algoritmi folositi

Programele care masoara timpii de executie a proceselor (alocarea de memorie, accesul la memorie, crearea threadurilor, thread context switch si thread migration) vor fi scrise in cele 3 limbaje de programare C, C++, Java, in Visual Studio, respectiv IntelliJ IDEA, iar interfata aplicatiei va fi implementata in Java. Utilizatorul va putea alege ce proces vrea sa testeze, iar pe interfata se vor afisa timpii de executie al acelui proces in toate cele 3 limbaje de programare si se va face o comparatie intre ele.

Pentru C si C++ :

- Timpul de execuție sau timpul CPU al unei sarcini date este definit ca timpul petrecut de sistem executând acea sarcină, adica timpul în care rulează un program. Există mai multe

moduri de a măsura timpul de execuție al unui program, precum functiile: time(), clock(), gettimeofday(), clock_gettime().[9]

time() : *time() function returns the time since the Epoch(jan 1 1970) in seconds.*
Header File : *"time.h"*
Prototype / Syntax : *time_t time(time_t *tloc);*
Return Value : *On success, the value of time in seconds since the Epoch is returned, on error -1 is returned. [9]*

clock() : *clock() returns the number of clock ticks elapsed since the program was launched.*
Header File : *"time.h"*
Prototype / Syntax : *clock_t clock(void);*
Return Value : *On success, the value returned is the CPU time used so far as a clock_t; To get the number of seconds used, divide by CLOCKS_PER_SEC.on error -1 is returned.[9]*

```
#include <bits/stdc++.h>

 using namespace std;

 void fun()      // A sample function whose time taken to be measured

 {   for (int i=0; i<10; i++) {...}

 }

 int main()

{       time_t start, end;  //  Returned time is in seconds

       // You can call it like this : start = time(NULL); in both the way start contain total time

          in seconds

        time(&start);

        ios_base::sync_with_stdio(false); // unsync the I/O of C and C++.

        fun();

        time(&end);  // Recording end time.

        double time_taken = double(end - start); // Calculating total time taken by the program.

        cout << "Time taken by program is : " << fixed<< time_taken << setprecision(5);

        cout << " sec " << endl;
```

return 0; } [9]

```
#include <bits/stdc++.h>

using namespace std;

void fun()  //A sample function whose time taken to be measured
{    for (int i=0; i<10; i++)  {…}
}

int main()
{
   /* clock_t clock(void) returns the number of clock ticks
      elapsed since the program was launched.To get the number
      of seconds used by the CPU, you will need to divide by
      CLOCKS_PER_SEC.where CLOCKS_PER_SEC is 1000000 on typical
      32 bit system.  */
   clock_t start, end;

   /* Recording the starting clock tick.*/
   start = clock();

   fun();

   // Recording the end clock tick.
   end = clock();

   // Calculating total time taken by the program.
   double time_taken = double(end - start) / double(CLOCKS_PER_SEC);
   cout << "Time taken by program is : " << fixed
        << time_taken << setprecision(5);
   cout << " sec " << endl;
   return 0;
} [9]
```

Am exemplificat cele doua moduri de masurare a timpilor de executie in C/C++ pe care le voi folosi (una dintre cele doua functii).

- Alocarea de memorie:

  Dinamica: ptr = (cast_type*) malloc(byte_size), ex: ptr=(int*) malloc(100*sizeof(int));

  Statica: int x; double d; char ch; int array[100];

- Accesul la memorie:

```
int MEMORY = 0;  // start at the beginning of memory
int location = 248 // set location of memory cell #248
MEMORY[ location ] = 65;      // save 65 in memory
int x = MEMORY[ location ];  // assign value from memory to x
```
[10]

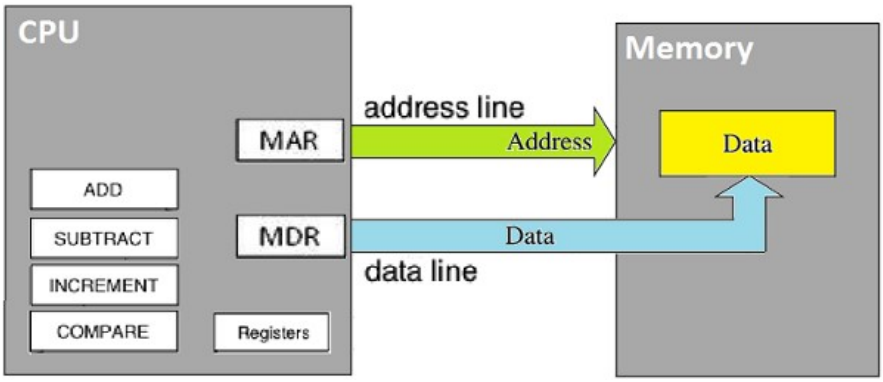


Figura 6 Memory access [10]

- Crearea unui thread:
  #include<pthread.h>
  pthread_t id[2];
  pthread_create(&id[0], NULL, printNumber, &arg); (id, atributele threadului, functia executata de thread, argumentele)

```c
// Global variable:
int i = 1;

// Starting routine:
void* foo(void* p){
    int i = *(int*) p;
    printf("Received value: %i", i);

    // Return reference to global variable:
    pthread_exit(&i);
}
```

```c
int i = 2;

void* foo(void* p){
  printf("%i\n", * (int*)p);

  pthread_exit(&i);
}

int main(void){
  pthread_t id;

  int j = 1;
  pthread_create(&id, NULL, foo, &j);

  int* ptr;
  pthread_join(id, (void**)&ptr);

  printf("%i\n", *ptr);
}
```

Figura 7 Crearea unui thread [11]

- Thread context switch:
  Să presupunem că există doar două procese, P1 și P2.
  P1 se execută și P2 așteaptă execuția. La un moment dat, sistemul de operare trebuie să schimbe P1 și P2, să presupunem că se întâmplă la a n-a instrucțiune a lui P1. Dacă t(x, k) indică marcajul de timp în microsecunde a instrucțiunii a k-a a procesului x, atunci schimbarea contextului ar lua t(2, 1) – t(1, n).[12]
  1.P2 se blochează în așteptarea datelor de la P1
  2.P1 marchează ora de începere.
  3.P1 trimite un token către P2.
  4.P1 încearcă să citească un token(jeton) de răspuns de la P2. Acest lucru induce un context switch.
  5.P2 este programat și primește tokenul.
  6.P2 trimite un token de răspuns către P1.
  7.P2 încearcă să citească un token de răspuns de la P1. Acest lucru induce un context switch.
  8.P1 este programat și primește tokenul.
  9.P1 marchează ora de încheiere.[12]
  Formula: $T = 2 * (Td + Tc + Tr)$

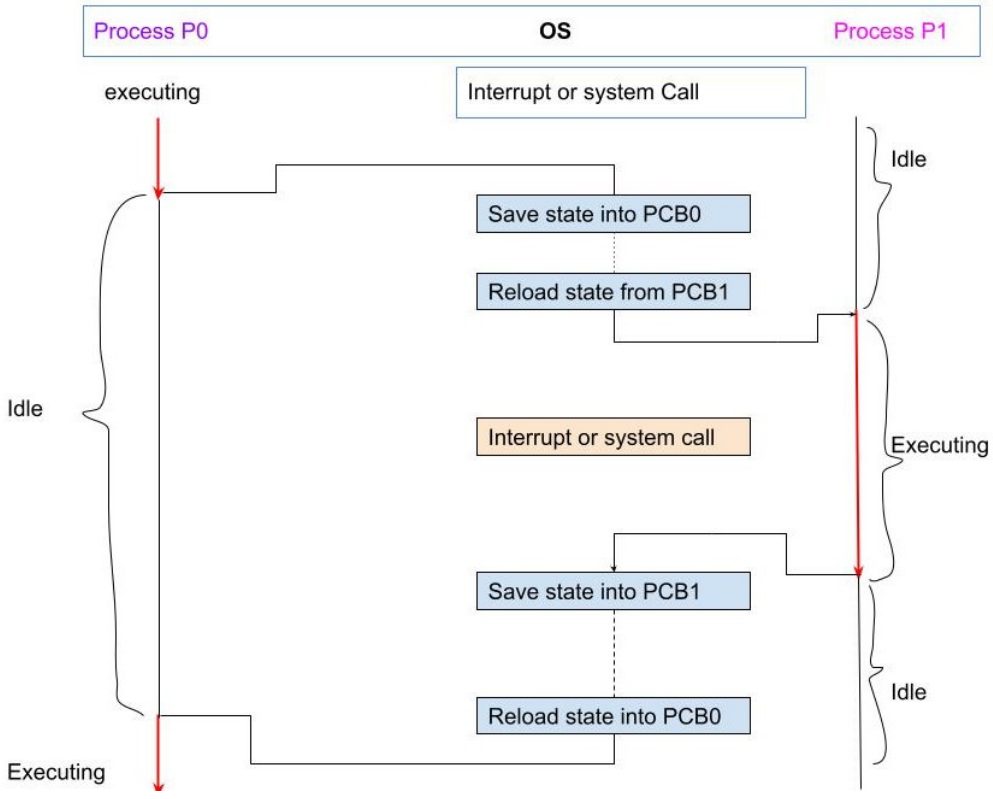


Figura 8 Context switch steps [13]

Pentru Java:

- Masurarea timpului de executie a proceselor se poate face prin mai multe metode, precum cu functiile: currentTimeMillis() –returneaza timpul in milisecunde, nanoTime().[14]

```
import java.time.Duration;
import java.time.Instant;
import org.apache.commons.lang3.time.StopWatch;
public class Example {
  public void test(){
    int num = 0;
    for(int i=0; i<=50; i++){
      num =num+i;
      System.out.print(num+", ");
    }
  }
  public static void main(String args[]){
    Example obj = new Example();
    long start1 = System.nanoTime();
    obj.test();
    long end1 = System.nanoTime();
    System.out.println("Elapsed Time in nano seconds: "+ (end1-start1));
    long start2 = System.currentTimeMillis();
    obj.test();
    long end2 = System.currentTimeMillis();
    System.out.println("Elapsed Time in milli seconds: "+ (end2-start2));
    Instant inst1 = Instant.now();
    obj.test();
    Instant inst2 = Instant.now();
    System.out.println("Elapsed Time: "+ Duration.between(inst1, inst2).toString());
    StopWatch stopWatch = new StopWatch();
    stopWatch.start();
    obj.test();
    stopWatch.stop();
    System.out.println("Elapsed Time in minutes: "+ stopWatch.getTime());
  }
}
```

Figura 9 Masurarea timpului de executie in Java [14]

- Alocarea de memorie:

Alocarea de memorie statică: În alocarea de memorie statică, trebuie să declarăm variabilele înainte de a executa programul. Memoria statică este alocată în timpul compilării.
Alocarea dinamică a memoriei: Alocarea dinamică a memoriei în Java înseamnă că memoria este alocată obiectelor Java în timpul de execuție sau în timpul execuției. Este contrar alocării statice de memorie. Alocarea dinamică a memoriei are loc în spațiul heap. Spațiul heap este locul în care obiectele noi sunt întotdeauna create și referințele lor sunt stocate în memoria stivei.[15]
Alocarea memoriei de stock în este utilizată pentru memoria statică și executia threadurilor. Valorile conținute în această memorie sunt temporare și limitate la metode specifice, deoarece continuă să fie menționate în modul Last-In-First-Out.
De îndată ce memoria este apelată și un nou bloc este creat în memoria stivei, memoria stivei păstrează apoi valori și referințe primitive cat timp dureaza metoda. După încheierea acestuia, blocul este șters și este disponibil pentru un nou proces. În general, dimensiunea totală a memoriei stivei este nesemnificativă față de cea a memoriei heap. Metodele folosite pentru alocarea memoriei stivei: push(), pop(), peek(), empty(), in search().[15]
Utilizat în principal de java runtime, Java Heap Space intră în joc de fiecare dată când un obiect este creat și alocat în el. Funcția discretă, precum Garbage Collection, continuă să elibereze memoria folosită de obiectele anterioare care nu au nicio referință. Pentru un obiect creat în Heap Space poate avea acces gratuit în aplicație. Alocarea memoriei în java este împărțită în părți, și anume Heap, Stack, Code și Static.[15]

```
package com.journaldev.test;

public class Memory {

        public static void main(String[] args) { // Line 1
                int i=1; // Line 2
                Object obj = new Object(); // Line 3
                Memory mem = new Memory(); // Line 4
                mem.foo(obj); // Line 5
        } // Line 9

        private void foo(Object param) { // Line 6
                String str = param.toString(); //// Line 7
                System.out.println(str);
        } // Line 8

}
```

Figura 10 Stack and Heap memory allocation [15]

- Crearea unui thread:

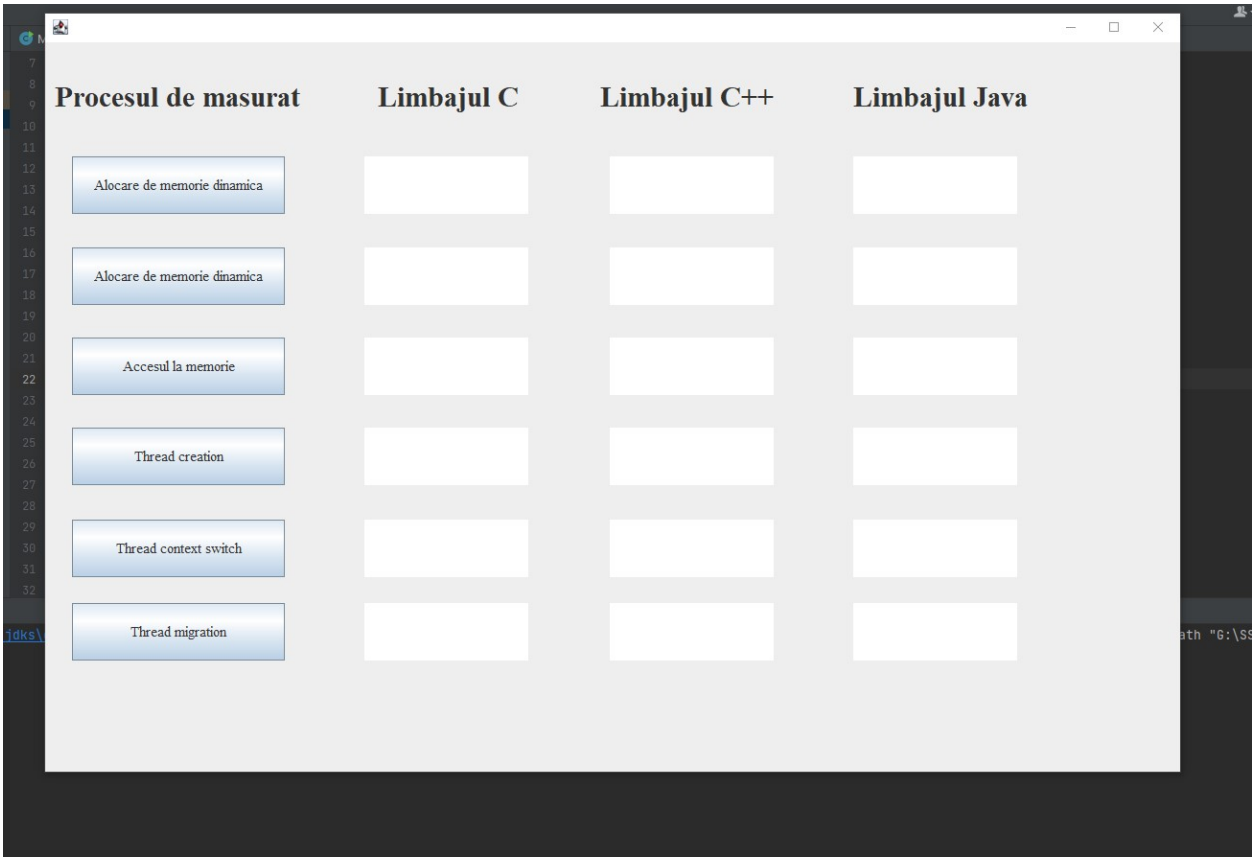
```
class Multi extends Thread{   //prin a extinde clasa Thread
public void run(){
System.out.println("thread is running...");
}
public static void main(String args[]){
Multi t1=new Multi();
t1.start();
 }
}  [16]
class Multi3 implements Runnable{  //prin a implementa interfata Runnable
public void run(){
System.out.println("thread is running...");
}
public static void main(String args[]){
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);   // Using the constructor Thread(Runnable r)
t1.start();
 }
} [16]
public class MyThread1  //folosind clasa Thread
{
// Main method
public static void main(String argvs[])
{
// creating an object of the Thread class using the constructor Thread(String name)
Thread t= new Thread("My first thread");

// the start() method moves the thread to the active state
t.start();
// getting the thread name by invoking the getName() method
String str = t.getName();
System.out.println(str);
}
} [16]
```

Procesele prezentate vor fi implementate, iar timpul lor de executie va fi masurat pentru fiecare limbaj de programare. Utilizatorul va folosi aplicatia pentru a afla timpii de executie a proceselor dorite si pentru a compara limbajele de programare, dorind sa afle care este mai eficient. El va porni programul, va afla timpul de executie a unui proces in cele trei limbaje de programare si le va compara.

4. Proiectare

Aplicatia va contine o interfata implementata in Java, care va primi executabilele programelor din Visual Studio (programe scrise in C si C++) si cel din IntelliJ IDEA (scris in Java), va lua valorile si va afisa utilizatorului rezultatul. Pentru fiecare proces (alocarea de memorie statica si dinamica, accesul la memorie, crearea unui thread, thread context switch si thread migration), voi avea un program diferit de masurare a timpului de executie. Utilizatorul va putea selecta procesul pe care doreste sa-l masoare (apasand un buton) si se va afisa timpul de executie a acelui proces in toate cele 3 limbaje de programare, pentru a putea fi comparate intre ele.



5. Implementare

Pentru realizarea acestui proiect, am implementat procesele: alocarea de memorie statica si dinamica, accesul la memorie static si dinamic, creare de thread-uri, thread context switch si thread migration in limbajele de programare C, C++ si Java, pentru a le masura timpul de executie si a le compara intre ele.

Pentru a masura timpul in C am folosit:

```
clock_t start, end;
double cpu_time_used;

start = clock();

end = clock();

cpu_time_used = (((double)(end - start)) / CLOCKS_PER_SEC);
```

Pentru a masura timpul in Java:

```
long start = System.nanoTime();

long end = System.nanoTime();
double difference = end - start;
double executionTime =difference/1000000;
```

Pentru alocarea de memorie dinamica si statica am alocat mai multe variabile dinamic si static si le am masurat timpul de executie. Pentru accesul la memorie static si dinamic am accesat variabile alocate dinamic si static si am masurat timpul de executie al accesului.

Acces la memorie dinamic:

```
long long int* x;
x = (long long int*)malloc(100 * sizeof(long long int));

p = x;
```

Acces la memorie static:

```
long long int* p;
long long int x;

p = &x;
```

```
int[] x = new int[1000];
double[] d= new double[1000];
```

Alocare de memorie dinamica:

```
int* ptr;
ptr = (int*)malloc(100 * sizeof(int));
```

Alocare de memorie statica:

```
int vec[10000];
long long int vec2[10000];
```

Pentru thread creation am creat mai multe threaduri si le am masurat timpul de executie.

```
pthread_t id;
pthread_t id2;
pthread_t id3;
pthread_create(&id, NULL, NULL, NULL);
pthread_create(&id2, NULL, NULL, NULL);
pthread_create(&id3, NULL, NULL, NULL);
```

```
Thread t1 = new Thread(thread1);
Thread t2 = new Thread(thread1);
Thread t3 = new Thread(thread1);
```

Pentru thread context switch am creat doua threaduri, am dat lock si unlock la lacat si apoi am mutat procesul de pe un thread pe altul, apoi am masurat timpul de executie al switch-ului de pe un thread pe altul.

```
void* thread_func1(void* arg)
{   pthread_mutex_lock(&mutex);
    pthread_mutex_unlock(&mutex);
    return NULL; }
void* thread_func2(void* arg)
{   pthread_mutex_lock(&mutex); }
```

```
public void func1() {
lock = new ReentrantLock();
```

```
lock.unlock();  }
public double func2() {
long start = System.nanoTime();
lock = new ReentrantLock();
long end = System.nanoTime(); }
```

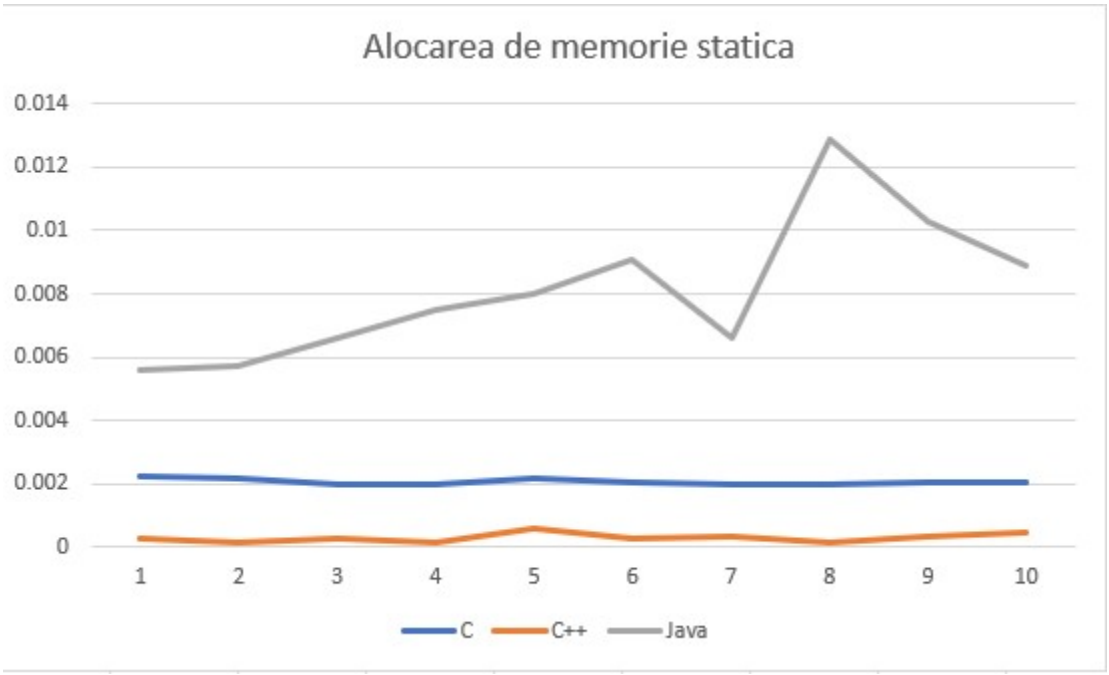
Pentru thread migration am creat 100 de threaduri, am luat thread-ul curent, am setat afinitatea procesorului (unde va migra threadul) si am trimis threadul de pe un core pe altul.

```
HANDLE thread = GetCurrentThread();
DWORD_PTR threadAffinityMask = (DWORD_PTR)arg;
BOOL success = SetProcessAffinityMask(thread, threadAffinityMask);
DWORD_PTR mask;
for(int i = 0; i < 4; i++){
   mask = 1 << i;
   pthread_create(&id[i], NULL, thread_func, (LPVOID)mask);   }
```

6. Testare si validare

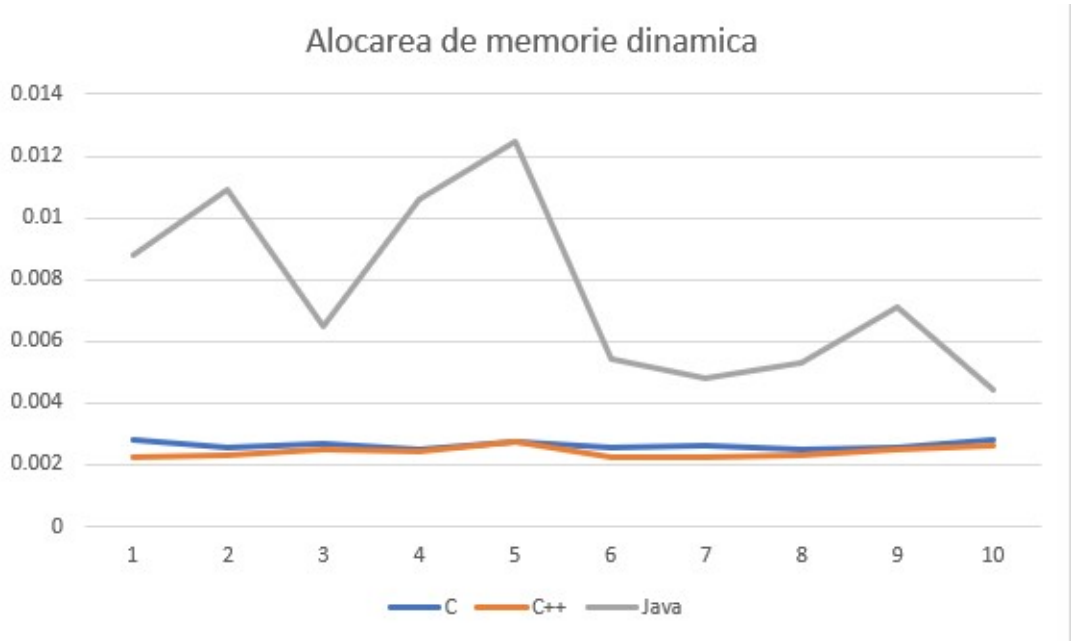
Pentru testare si validare, am rulat fiecare program, fiecare proces separat pentru cele trei limbaje de programare C, C++ si Java, am salvat masuratorile intr-un fisier, am luat datele din fisier si le-am introdus intr-un tabel, apoi am generat un grafic si am comparat graficele intre ele.

Alocarea de memorie statica:			
Limbajul:	C	C++	Java
test1:	0.002214286	0.000285714	0.0056
test2:	0.002157143	0.000142857	0.0057
test3:	0.001971429	0.000285714	0.0066
test4:	0.001985714	0.000142857	0.0075
test5:	0.002185714	0.000571429	0.008
test6:	0.002028571	0.000285714	0.0091
test7:	0.002	0.000342847	0.0066
test8:	0.001971429	0.000142857	0.0129
test9:	0.002028571	0.000347645	0.0103
test10:	0.002057143	0.000424227	0.0089



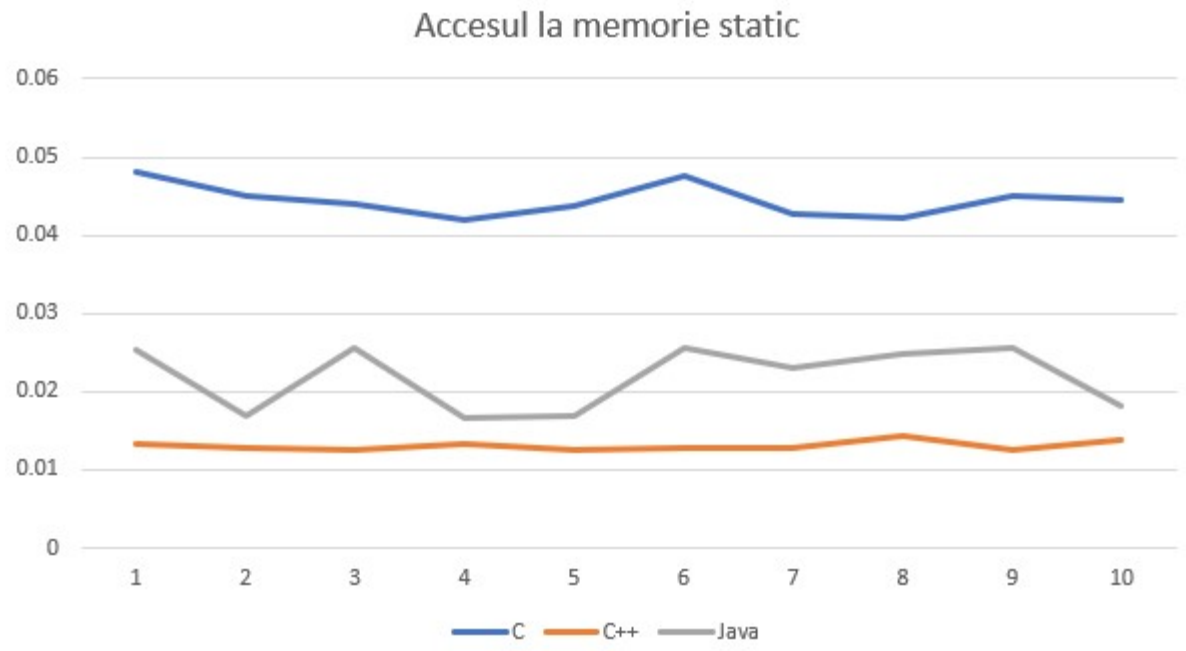
Din primul grafic putem compara timpii de executie a alocarii de memorie statica si putem observa ca in C++ se face cel mai rapid, dupa care in C, iar apoi in Java.

Alocarea de memorie dinamica:			
Limbajul:	C	C++	Java
test1:	0.00279	0.00225	0.0088
test2:	0.00256	0.00234	0.0109
test3:	0.0027	0.00251	0.0065
test4:	0.00251	0.00244	0.0106
test5:	0.00275	0.00275	0.0125
test6:	0.00253	0.00227	0.0054
test7:	0.0026	0.00223	0.0048
test8:	0.00249	0.00231	0.0053
test9:	0.00259	0.00249	0.0071
test10:	0.00283	0.00263	0.0044



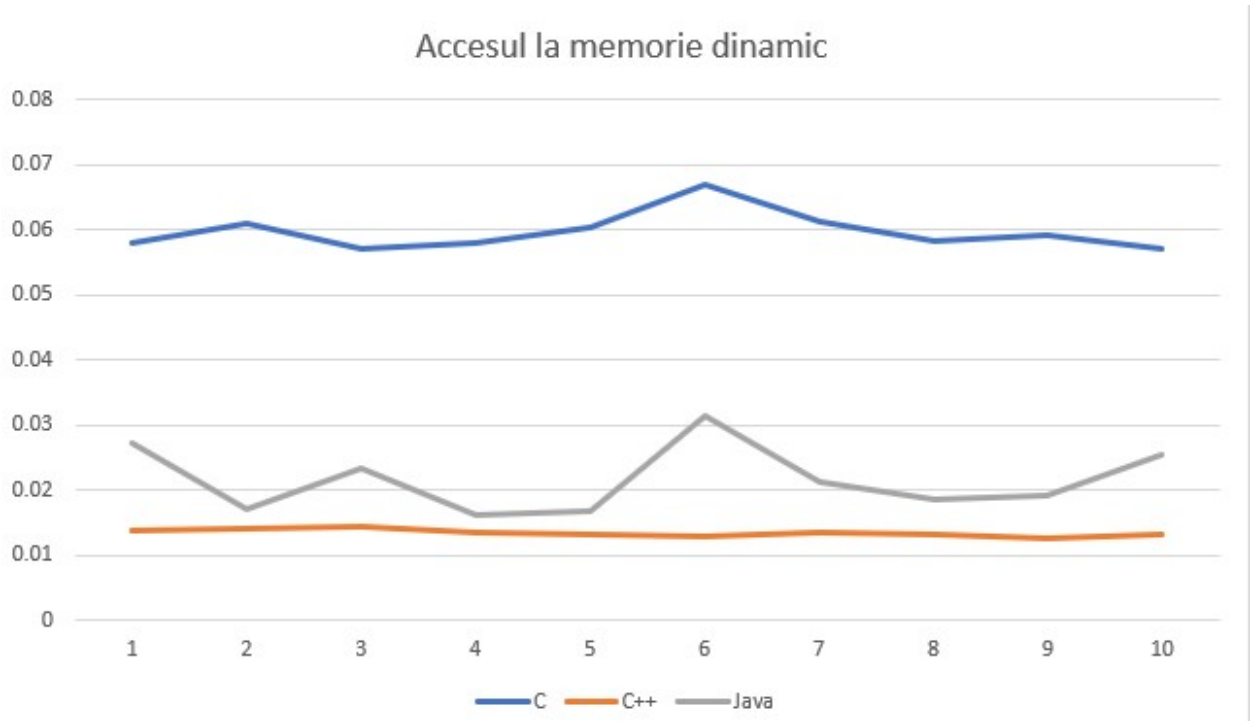
Din al doilea grafic putem compara timpii de executie a alocarii de memorie dinamica si putem observa ca in C++ se face cel mai rapid, dupa care in C, iar apoi in Java, in C si C++ timpii fiind foarte apropiati.

Accesul la memorie static:			
Limbajul:	C	C++	Java
test1:	0.048	0.0134	0.0253
test2:	0.045	0.0128	0.0168
test3:	0.044	0.0125	0.0257
test4:	0.042	0.0133	0.0167
test5:	0.04366	0.0126	0.017
test6:	0.0476	0.0129	0.0257
test7:	0.04266	0.0127	0.0231
test8:	0.04233	0.0144	0.0249
test9:	0.045	0.0126	0.0255
test10:	0.0446	0.0138	0.0183



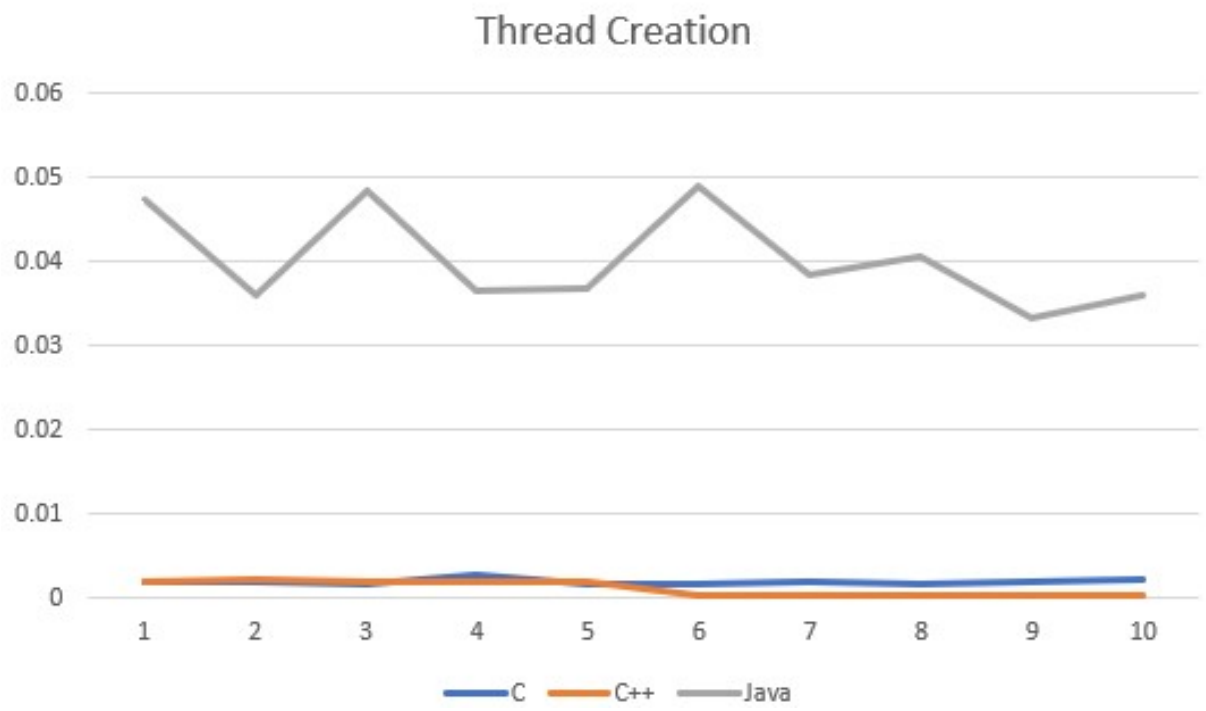
Din al treilea grafic putem compara timpii de executie a accesului la memorie static si putem observa ca in C++ se face cel mai rapid, dupa care in Java, iar apoi in C.

Accesul la memorie dinamic:			
Limbajul:	C	C++	Java
test1:	0.058	0.0138	0.0273
test2:	0.061	0.0141	0.0171
test3:	0.057	0.0143	0.0234
test4:	0.058	0.0134	0.0163
test5:	0.06033	0.0131	0.0169
test6:	0.067	0.0129	0.0313
test7:	0.0613	0.0135	0.0212
test8:	0.05833	0.0133	0.0187
test9:	0.05933	0.0127	0.0193
test10:	0.057	0.0132	0.0255



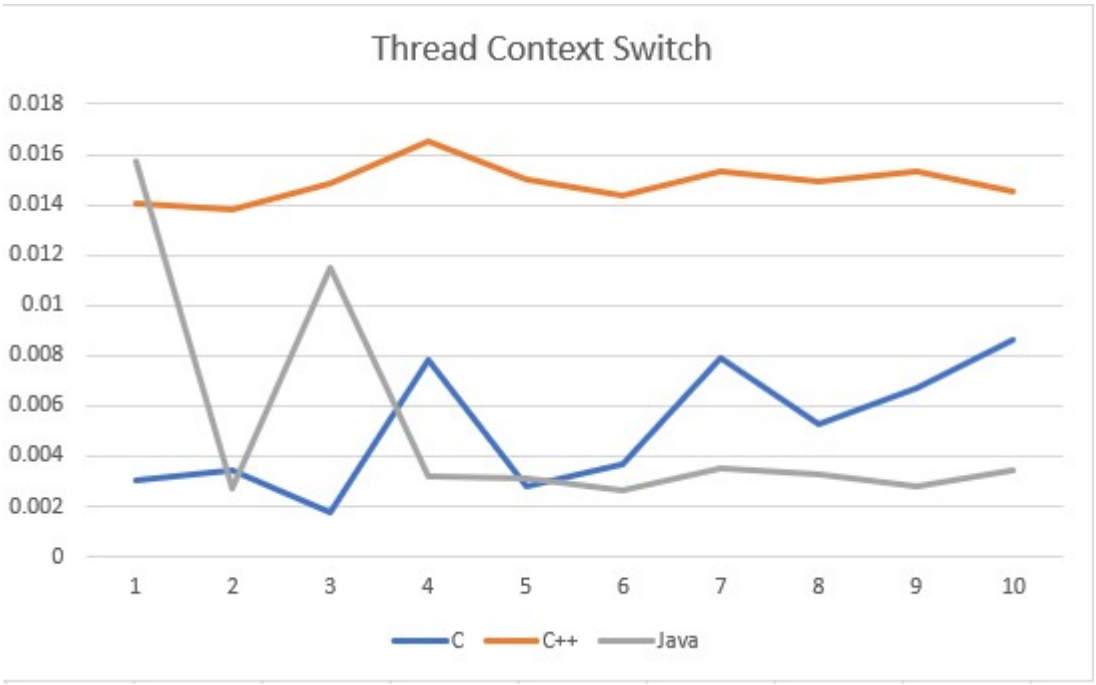
Din al patrulea grafic putem compara timpii de executie a accesului la memorie dinamic si putem observa ca in C++ se face cel mai rapid, dupa care in Java, iar apoi in C.

Thread Creation:			
Limbajul:	C	C++	Java
test1:	0.002046	0.001784	0.0473
test2:	0.001796	0.002179	0.0359
test3:	0.001708	0.001901	0.0484
test4:	0.00268	0.0018	0.0366
test5:	0.00175	0.001815	0.0368
test6:	0.001729	0.0002189	0.0489
test7:	0.001806	0.0001842	0.0384
test8:	0.001734	0.0001934	0.0407
test9:	0.001962	0.0001913	0.0333
test10:	0.002058	0.0001811	0.0361



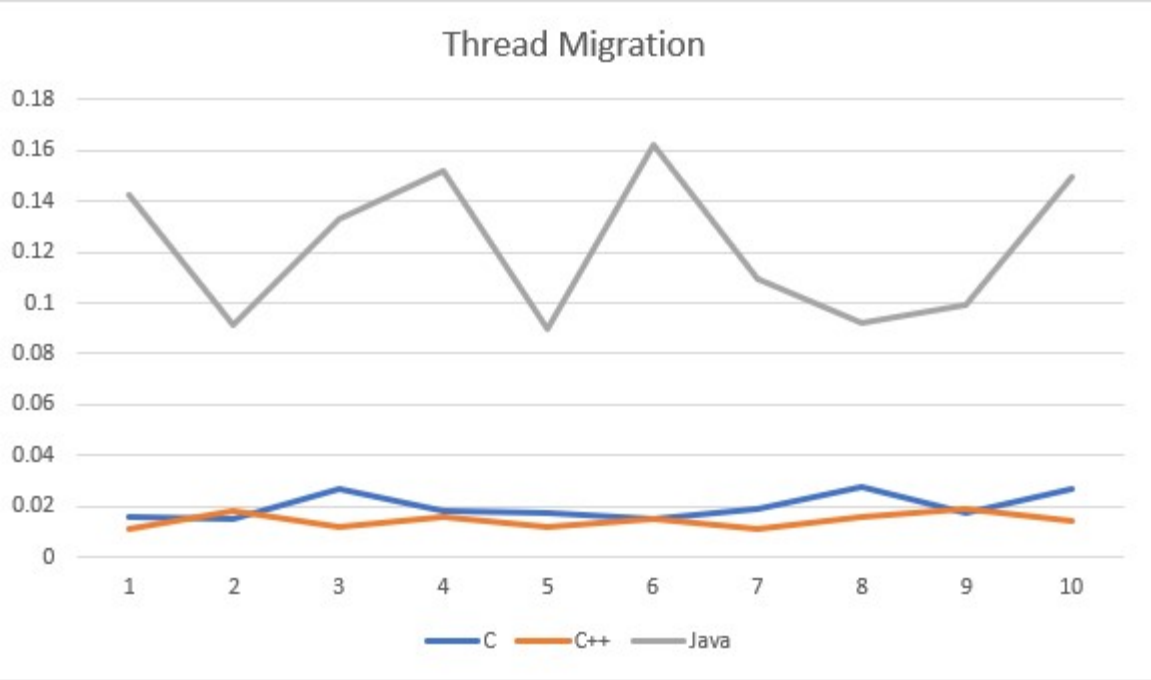
Din al cincilea grafic putem compara timpii de executie a crearii de threaduri si putem observa ca in C++ se face cel mai rapid, dupa care in C, iar apoi in Java.

Thread Context Switch:			
Limbajul:	C	C++	Java
test1:	0.003	0.014046	0.0157
test2:	0.0034	0.0137869	0.0027
test3:	0.00175	0.0148831	0.0115
test4:	0.0078	0.0165343	0.0032
test5:	0.0028	0.0149957	0.0031
test6:	0.00366	0.0143885	0.0026
test7:	0.0079	0.0153173	0.0035
test8:	0.00524	0.0149345	0.0033
test9:	0.0067	0.0153173	0.0028
test10:	0.0086	0.0145491	0.0034



Din al saselea grafic putem compara timpii de executie a thread context switch si putem observa ca in C se face cel mai rapid, dupa care in Java, iar apoi in C++.

Thread Migration:			
Limbajul:	C	C++	Java
test1:	0.016	0.011	0.1429
test2:	0.015	0.018	0.0916
test3:	0.027	0.012	0.1328
test4:	0.018	0.016	0.1521
test5:	0.017	0.012	0.0902
test6:	0.015	0.015	0.1626
test7:	0.019	0.011	0.1092
test8:	0.028	0.016	0.0922
test9:	0.017	0.019	0.0994
test10:	0.027	0.014	0.1496



Din al saptelea grafic putem compara timpii de executie a thread migration si putem observa ca in C++ se face cel mai rapid, dupa care in C, iar apoi in Java.

7. Concluzii

In concluzie, prin acest proiect am reusit sa masor timpii de executie a unor procese (alocarea de memorie statica si dinamica, accesul la memorie static si dinamic, crearea threadurilor, thread context switch si thread migration) in trei limbaje de programare C, C++ si Java. Aceste masuratori le-am pus in tabele si am generat niste grafice pentru a putea face o comparatie mai buna intre timpii de executie a unui proces in toate cele trei limbaje de programare. Vizualizand graficele, putem observa care limbaj este mai eficient pentru un anumit proces, avand timpul de executie mai mic.

8. Bibliografie

[1] How to measure execution time of a program: https://serhack.me/articles/measurexecutiontime-program/

[2] Memory Allocation: https://www.cs.uah.edu/~rcoleman/Common/C_Reference/MemoryAlloc.html

[3] Geeksforgeeks, Memory Access Methods: https://www.geeksforgeeks.org/memory-access-methods/

[4] Techopedia, Direct Memory access: https://www.techopedia.com/definition/2767/direct-memory-access-dma

[5] Wikipedia, Thread: https://en.wikipedia.org/wiki/Thread_(computing)

[6] How to create a simple thread: https://www.educative.io/answers/how-to-create-a-simple-thread-in-c

[7] Geeksforgeeks: https://www.geeksforgeeks.org/difference-between-thread-context-switch-and-process-context-switch/

[8] Quora: https://www.quora.com/What-is-thread-migration-Should-the-thread-image-be-the-same-moving-from-one-process-to-another

[9] Geeksforgeeks, Measure execution time with high precision in C/C++: https://www.geeksforgeeks.org/measure-execution-time-with-high-precision-in-c-c/

[10] Memory access: http://www.c-jump.com/bcc/c155c/MemAccess/MemAccess.html

[11] How to create a simple thread in C: https://www.educative.io/answers/how-to-create-a-simple-thread-in-c

[12] Geeksforgeeks, Measure the time spent in context switch: https://www.geeksforgeeks.org/measure-time-spent-context-switch/

[13] Measure the time spent in context switch: https://www.codingninjas.com/codestudio/library/measure-the-time-spent-in-context-switch

[14] How to measure execution time for a Java method: https://www.tutorialspoint.com/how-to-measure-execution-time-for-a-java-method

[15] Memory allocation in Java: https://www.upgrad.com/blog/memory-allocation-in-java/

[16] How to create a thread in Java: https://www.javatpoint.com/how-to-create-a-thread-in-java