Documentatie

Macovei Alexandru-Fabian

1 Organizarea informatiei

Informatia este organizata pe 5 directoare: **Implementation**, **Random_data_generator**, **Unit_test**, **Data** si **Documentation**. Directorul **Implementation** contine mai multe fisiere cu implementarea algoritmilor de planificare si cu functii auxiliare necesare. Folderul **Random_data_generator** contine generatorul de date aleatoare parametrizat. Directorul **Unit_test** contine fisierele cu datele de test si cu implementarea metodei de testare. Folderul **Data** contine fisierele cu date introduse manual si cu date generate automat. Directorul **Documentation** contine cateva fisiere **.pdf** suport cu detalii despre proiect, printre care si acesta. Codul din fisiere a fost organizat in asa fel incat nici un fisier header sa nu fie inclus de mai multe ori.

1.1 Structurile

In fisierele **Implementation/structs.h** si **Implementation/structs.cpp** au fost declarate, respectiv implementate structurile de baza regasite pe tot parcursul proiectului:

1. structura **process** contine atributele unui proces. Campurile structurii:
   (a) **pid** - un numar intreg reprezentand id-ul unic al procesului
   (b) **time** - nu numar intreg reprezentand timpul necesar executiei procesului (in **[ms]**)
   (c) **priority** - un numar intreg reprezentand prioritatea procesului in clasa sa de prioritate
   (d) **priority_class**
   - un numar intreg reprezentand clasa de prioritate a procesului
   - pe parcursul implementarii s-au utilizat 4 clase de prioritate descrise prin numere de la 0 la 3, dar acest lucru poate fi modificat cu destul de multa usurinta
2. structura **cpu_burst** contine atributele unei instante in profilul de executie. Campurile structurii:
   (a) **start_time** - un numar intreg reprezentand timpul de inceput al unei instante (in **[ms]**)

(b) **stop_time** - un numar intreg reprezentand timpul de sfarsit al unei instante (in [**ms**])

(c) **duration** - un numar intreg reprezentand durata unei instante (in [**ms**])

(d) **pid** - un numar intreg reprezentand id-ul procesului care a fost executat in instanta de executie respectiva

Aceste fisiere sunt include in **Fisierele utility**

1.2 Fisierele utility

Clasa **utility** isi are interfata si implementarea in fisierele **Implementation/utility.h**, respectiv **Implementation/utility.cpp** si contine 3 metode statice cu topul returnat void:

1. **read_process_queue(std::deque<process>& queue, const char* file)**
   - metoda care citeste o coada de procese dintr-un fisier ales
   - utilizarea unui **deque** in implementarea algoritmilor a fost o alegere facuta deoarece **deque** are aceleasi functionalitati ca **vector**, dar este putin mai eficient pentru inserari. Cu toate acestea se putea utiliza la fel de bine si o structura **vector**

2. **reset_execution_data**
   - metoda care reseteaza valorile datelor din profilul de executie al algoritmului

3. **print_tuple_list**
   - metoda care afiseaza profilul de executie sub forma de lista de tuple - in interiorul acestei functii se va apela si **reset_execution_data**, descrisa anterior

In afara clasei **utility**, fisierul **Implementation/utility.h** mai contine si 6 expresii constante reprezentand numele care va aparea in dreptul fiecarui algoritm la apelarea metodei **print_tuple_list**. Aceste fisiere sunt incluse in **Fisierele cu algoritmi**

1.3 Fisierele cu algortmii de planificare

Algoritmii de planificare isi au antetul in **Implementation/algorithms.h** si implementarea in **Implementation/algorithms.cpp**. Aceste fisiere contin o macrocomanda, **arguments**, care se expandeaza in parametrii formali pe care ii au toti algoritmii de planificare, si o clasa **algorithms**, care implementeaza algoritmii, cu urmatorii membrii statici:

1. **nr_of_levels** - o constanta intreaga care retine cu cate clase de prioritati se lucreaza

2. **void(*(functions[]))(arguments, int bonus_process_count)** - un vector de pointeri catre functii care arata ce algoritmi se vor utiliza pentru fiecare clasa de prioritate in cazul algoritmului **Multi_level_Queue_Scheduling**

3. **FCFS** - metoda care implementeaza algoritmul **FCFS**

4. **SJF** - metoda care implementeaza algoritmul **SJF**

5. **RR** - metoda care implementeaza algoritmul **Round-Robin**

6. **Priority_Scheduling** - metoda care implementeaza algoritmul **Priority scheduling**

7. **RR_Priority_Scheduling** - metoda care implementeaza algoritmul **Round-Robin Priority Scheduling**

8. **Multi_level_Queue_Scheduling** - metoda care implementeaza algoritmul **Multi-level Queue Scheduling**

Specificatii privind implementarea acestor algoritmi se gaseste in comentariile codului sursa. Aceste fisiere sunt incluse in **Fisierele de teste**.

1.4 Folderul de teste

Contine 2 directoare: **Inputs** cu 2 fisiere (**test1.txt** si **test2.txt**) care contin datele de intrare pentru teste, si **Tests** cu 2 fisiere (**tests.h** si **tests.cpp**). Fisierul **tests.h** contine 2 macrocomenzi ce se expandeaza in bucati de cod ce compara a rezultatul obtinut in urma executarii fiecarui algoritm pe datele de intrare din **test1.txt** si **test2.txt** cu rezultatul corect, introdus la mana in metoda de testare. Metoda de testare isi are antetul in **tests.h** si implementarea in **tests.cpp** si testeaza pe rand fiecare algoritm executat pe datele de intrare din fiecare fisier de intrare.

Acest cod este inclus in fisierul **Implementation/main.cpp**.

1.5 Generatorul de date aleatoare

Implementeaza o clasa ce contine un generator de numere aleatoare (**mt19937**), o distributie uniforma a datelor si o distributie normala a datelor. Metodele private ale clasei **random_data_generator**:

1. **random_data_generator()**
   - constructorul clasei - instantiaza un obiect al clasei conform parametrilor specificati sau cu parametrii impliciti
   - obiectul este folosit mai apoi pentru a genera un intreg set de date
   - constructorul este privat pentru a pastra principiul incapsularii datelor

2. **generate_uniform()** - genereaza un numar dupa o distributie uniforma parametrizata in constructor

3. **generate_normal()** - genereaza un numar dupa o distributie normala parametrizata in constructor

Pe langa metodele private, clasa are o metoda statica publica (**data_set_generator()**), ce foloseste metodele private pentru a genera un intreg set de date in fisier. Aceasta este o metoda cu parametrii cu valori implicite modificabile si este singura metoda care se apeleaza in alte fisiere ale programului.

1.6 Fisierul main

Implementeaza un meniu cu ajutorul a 2 instructiuni **switch** intr-o bucla infinita. Pe langa metodele de rulare a testelor cu datele introduse manual sau generate aleator, Fisierul mai contine cateva functiile:

1. **input_int** - intoarce un numar intreg citit de la tastatura dupa ce trateaza posibile introduceri neadecvate de date (ex: un caracter)

2. **input_custom_parameters** - citeste 5 numere intregi de la tastatura, numere ce vor fi trimise ca parametrii de generare aleatoare a datelor

2 Utilizarea programului

2.1 Instructiuni de utilizare

Folderul **Proiect** contine toate directoarele si fisierele prezentate anterior si inca un fisier in plus: fisierul Makefile. Acesta este facut pentru a compila tot ce e necesar deodata.

In acest sens, un apel al comenzii **make** in **Command Prompt** odata ce directorul in care ne aflam este directorul **Proiect** este suficient pentru a compila toate fisierele. Pentru a sterge apoi fisierele **.o** si **.exe** din directorul **Proiect** se va apela comanda **make clean**.

Odata ce fisierul **main.exe** a fost creat, acesta poate fi rulat cu o comanda **main** in **Command Prompt** (sau **./main** in **Power Shell**). In consola va aparea meniul de la care pleaca selectia oricarei functionalitati ale programului. Optiunile se aleg introducand numarul intreg corespunzator optiunii alese (aflat in fata descrierii optiunii respective), si respectiv apasand teste **Enter**.

Datele de intrare introduse manual se pun in fisierul **Data/manual_data.txt** sub forma tabelara astfel:

1. Toate datele sunt numere intregi.

2. Pe prima linie, separat de restul tabelului, se afla numarul de procese care vor fi citite. Acest numar trebuie sa corespunda cu numarul de linii din tabel.

3. Pe prima coloana se afla **pid** (id-urile fiecarui proces). Pentru o intelegere mai buna a rezultatelor se recomanda ca **pid** sa fie unice, dar aceasta conditie nu este necesara

4. Pe a 2-a coloana sunt clasele de prioritate ale fiecarui proces. Ele trebuie sa fie numere intregi in intervalul [0, **algorithms::nr_of_levels** − 1] (implicit intervalul este [0, 3]).

5. A 3-a coloana contine prioritatile proceselor.

6. Pe ultima coloana se afla timpul de executie al fiecarui proces.

2.2 Exemplu de utilizare

Pentru compilare este necesar compilatorul de **C++ GNU**, **g++**.

```
C:\Proiect>make
g++ -Wall -c Implementation/structs.cpp -o structs.o
g++ -Wall -c Implementation/utility.cpp -o utility.o
g++ -Wall -c Implementation/algorithms.cpp -o algorithms.o
g++ -Wall -c Unit_test/Tests/tests.cpp -o tests.o
g++ -Wall -c Random_data_generator/random_data_generator.cpp -o random_data_generator.o
g++ -Wall -c Implementation/main.cpp -o main.o
g++ -Wall structs.o utility.o algorithms.o tests.o random_data_generator.o main.o -o main.ex

C:\Proiect>main
Select an option:
   1. Run keyboard input data (from Data/manual_data.txt)
   2. Run randomly generated data (generated to Data/random_data.txt)
   3. Run tests (from Unit test/Inputs/test1.txt and Unit test/Inputs/test2.txt)
   4. Exit

1

/*output*/

Select an option:
   1. Run keyboard input data (from Data/manual_data.txt)
   2. Run randomly generated data (generated to Data/random_data.txt)
   3. Run tests (from Unit test/Inputs/test1.txt and Unit test/Inputs/test2.txt)
   4. Exit

2

Select an option:
   1. Run with default parameters (10, 1, 9, 0, 7)
   2. Run with custom parameters
   3. Return to the previous menu

1
```

```
/*output*/

Select an option:
   1. Run keyboard input data (from Data/manual_data.txt)
   2. Run randomly generated data (generated to Data/random_data.txt)
   3. Run tests (from Unit test/Inputs/test1.txt and Unit test/Inputs/test2.txt)
   4. Exit

2

Select an option:
   1. Run with default parameters (10, 1, 9, 0, 7)
   2. Run with custom parameters
   3. Return to the previous menu

2

Input the number of processes to generate:
10

Input the minimum priority:
1

Input the maximum priority:
19

Input the mean of the normal distribution for the burst times:
0

Input the deviation of the normal distribution for the burst times:
14

/*output*/

Select an option:
   1. Run keyboard input data (from Data/manual_data.txt)
   2. Run randomly generated data (generated to Data/random_data.txt)
   3. Run tests (from Unit test/Inputs/test1.txt and Unit test/Inputs/test2.txt)
   4. Exit

3

/*output*/

Select an option:
   1. Run keyboard input data (from Data/manual_data.txt)
```

```
   2. Run randomly generated data (generated to Data/random_data.txt)
   3. Run tests (from Unit test/Inputs/test1.txt and Unit test/Inputs/test2.txt)
   4. Exit

4


C:\Mac\Facultate\SO\Teme\Proiect>make clean
del main.exe *.o
```

3 Probleme intalnite

3.1 Discrepante intre compilatoare

Initial proiectul a fost facut in **Visual Studio 2022**, folosind compilatorul **Microsoft C++ Compiler** (**MSVC**) si pentru clasa **random_data_generator** mai era utilizat un membru care seta seed-ul generatorului, **std::random_device**. Acesta genera un numar aleator dintr-o sursa nedeterminista (de exemplu entropia unei componente hardware). La trecerea de la **MSVC** la compilatorul **GNU g++** am observat ca datele generate erau aceleasi la fiecare rulare a functiei de generare. Am facut, din acest motiv, trecerea de la **random_device** la **time(0)** pentru setarea valorii initiale pentru generator, solutie care a rezolvat problema.

3.2 Lucrul cu o structura mai complexa de fisiere

Din cauza faptului ca nu am mai lucrat cu atat de multe fisiere in diferite directoare, una dintre greutati a fost decizia cu privire la locul includerilor. Alte probleme au fost cauzate de includeri circulare, pe care le-am rezolvat mutad variabile globale intre fisiere astfel incat sa elimin una dintre includeri.

3.3 Pointeri catre functii

O dificultate mai mica a reprezentat-o vectorul de pointeri catre functii. Din cauza faptului ca nu am mai utilizat pointeri catre functii pana acum mi-a fost mai greu sa inteleg cum functioneaza acestia, dar plecand de la niste exemple mai simple si testand ipoteze proprii cu privire la modul de utilizare am reusit sa implementez si sa utilizez cu succes acel vector.

3.4 Fisierul Makefile

Pentrul ca nu am mai alcatuit un fisier **Makefile** de la inceput pana acum, nu este de mirare faptul ca nu a fost cea mai usoara sarcina pe care a trebuit sa o indeplinesc. Am avut devoie de mai multe exemple pentru cazuri mai simple de structuri de fisiere pentru a intelege cum ar trebui sa arate **Makefile-ul** programului curent. Sunt multumit cu rezultatul final al fisierului, dar nu pot

spune cu certitudine ca este cea mai buna modalitate de a-l scrie data fiind lipsa mea de experienta.