Palestine Polytechnic University

Operating Systems

Final Project (phase 1 & 2)

Course instructor:

Dr. Radwan Tahbob

Prepared by:

Samir Abuisneneh

Phase 1:

Simulate 3 different process scheduling algorithms (draw a gantt chart and get all important information: Wait Time, Finish Time, Turnaround Time)

Scheduling Algorithms:

First Come First Serve:

The simplest scheduling algorithm. FCFS simply queues processes in the order that they arrive in the ready queue.

In this, the process that comes first will be executed first and the next process starts only after the previous gets fully executed.

Code implementation:

To work with FCFS the program uses the function `void FCFS()`

The code for FCFS is fairly simple, it takes a vector of PCB blocks, sorts them based on their arrival time using the

```
bool compareTime(PCB a, PCB b) // to sort the FCFS and RR vector
based on arrival time

{

   return a.arTime < b.arTime;

}
```

The function works by comparing the arrival time part of the PCB and choosing the smaller option.

After sorting the PCB blocks a counter starts called (progress) which is incremented after each addition to the gantt chart, and it also checks if this is the last element, if not if adds the context switch to the progress

The progress is used to calculate finish time and wait time

```
waitTime.push_back(progress - processFCFS[i].arTime);

finishTime.push_back(progress);
```

Example Output:

```
/\/\/\/\/\/\/\/\/\ FCFS /\/\/\/\/\/\/\/\/\
x----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x
|P1  |CS   |P2   |CS   |P0   |CS   |P3   |CS   |P4   |CS   |P5   |CS   |P6   |
x----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x
|0 12|13 14|13 16|17 18|17 27|28 29|28 49|50 51|50 57|58 59|58 63|64 65|64 76|
x----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x

Process 1
Finish Time = 12
Wait Time = 0
Turnaround Time = 12

Process 2
Finish Time = 16
Wait Time = 12
Turnaround Time = 15

Process 0
Finish Time = 27
Wait Time = 14
Turnaround Time = 24

Process 3
Finish Time = 49
Wait Time = 23
Turnaround Time = 44

Process 4
Finish Time = 57
Wait Time = 41
Turnaround Time = 48

Process 5
Finish Time = 63
Wait Time = 46
Turnaround Time = 51

Process 6
Finish Time = 76
Wait Time = 48
Turnaround Time = 60

Average Finish time = 42.8571
Average waiting time = 26.2857
Average Turnaround time = 36.2857
CPU Utilization = 92.1053%
```

Shortest Job First:

It's an algorithm in which the process having the smallest execution time is chosen for the next execution. This scheduling method can be preemptive or non-preemptive. It significantly reduces the average waiting time for other processes awaiting execution.

Thanks to Mr. Omar Aburish for providing me with the idea for this implementation

Code Implementation:

First the program checks what processes have arrived (using the elapsed time), and it adds them to a vector called (ready) and it removes them from the original PCB vector which is called (processSJF)

Then the program sorts the ready vector based on the shortest CPU burst time using

```
bool compareCPUburst(PCB a, PCB b) // to sort the SJF vector based on CPU burst time

{

   return a.CPUburst < b.CPUburst;

}
```

Which returns the shortest process first

Then the top most process of the ready vector is inserted into the (execOrder) vector which marks the order of execution, while this is happening the elapsed time is increased with the CPU burst of the chosen process

```
progress += execOrder[i].CPUburst;
```

Wait and Finish time are calculated the same as FCFS

```
waitTime.push_back(progress - execOrder[i].arTime);

finishTime.push_back(progress);
```

Example Output:

```
/\/\/\/\/\/\/\/\/\/\ FCFS /\/\/\/\/\/\/\/\/\/\
x----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x
|P1  |CS   |P2   |CS   |P0   |CS   |P3   |CS   |P4   |CS   |P5   |CS   |P6   |
x----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x
|0 12|13 14|13 16|17 18|17 27|28 29|28 49|50 51|50 57|58 59|58 63|64 65|64 76|
x----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x

Process 1
Finish Time = 12
Wait Time = 0
Turnaround Time = 12

Process 2
Finish Time = 16
Wait Time = 12
Turnaround Time = 15

Process 0
Finish Time = 27
Wait Time = 14
Turnaround Time = 24

Process 3
Finish Time = 49
Wait Time = 23
Turnaround Time = 44

Process 4
Finish Time = 57
Wait Time = 41
Turnaround Time = 48

Process 5
Finish Time = 63
Wait Time = 46
Turnaround Time = 51

Process 6
Finish Time = 76
Wait Time = 48
Turnaround Time = 60

Average Finish time = 42.8571
Average waiting time = 26.2857
Average Turnaround time = 36.2857
CPU Utilization = 92.1053%
```

Round Robin:

Round Robin is a CPU scheduling algorithm where each process is assigned a fixed time slot in a cyclic way called a quantum.

Code Implementation:
First the program calculates the number of iterations needed using since there maybe more iterations than processes if ones CPU burst exceeds the Quantum

```
for (int i = 0; i < processRR.size(); i++) //calculate the number of iterations to finish all processes
   {
       double temp = processRR[i].CPUburst / 10.0;
       numOfIterations += ceil(temp);
   }
```

Then the program looks for processes that have arrived and it sorts them based on their arrival time like FCFS

```
if (processRR[j].arTime <= elapsed) //check if process has arrived
     {
       ready.push_back(processRR[j]);
       sort(ready.begin(), ready.end(), compareTime); //sort on CPU burst
       processRR.erase(processRR.begin() + j); //remove selected item from original vector
       j--;
     }
```

Then it inserted the first arriving process into the round robin queue.
To increase the progress (elapsed time) it checks if the CPU burst is larger than the quantum

```
if (temp.CPUburst - quantum > 0) //if CPU burst of process is larger than quantum move progress by counter value
    {
```

```
            progress += quantum;
        }
            else
        {
            progress += temp.CPUburst;
        }
```

Then it subtracts the quantum form the process's CPU burst

```
temp.CPUburst -= quantum; //subtract quantum from CPU burst
```

To calculate the wait and finish time:

```
waitTime[temp.processID] += abs(progress -
finishTime[temp.processID]); //wait time for each process
finishTime[temp.processID] = progress;
```

Example Output:

```
/\/\/\/\/\/\/\/\/\ RoundRobin /\/\/\/\/\/\/\/\/\
x----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x
|P1  |CS   |P2   |CS   |P0   |CS   |P3   |CS   |P4   |CS   |P1   |CS   |P5   |CS   |P6   |CS   |P3   |CS   |P6   |CS   |P3   |
x----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x
|0 10|10 11|11 14|14 15|15 25|25 26|26 36|36 37|37 44|44 45|45 47|47 48|48 53|53 54|54 64|64 65|65 75|75 76|76 78|78 79|79 79|
x----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x

Process 0
Finish Time = 25
Wait Time = 12
Turnaround Time = 22

Process 1
Finish Time = 47
Wait Time = 35
Turnaround Time = 47

Process 2
Finish Time = 14
Wait Time = 10
Turnaround Time = 13

Process 3
Finish Time = 79
Wait Time = 53
Turnaround Time = 74

Process 4
Finish Time = 44
Wait Time = 28
Turnaround Time = 35

Process 5
Finish Time = 53
Wait Time = 36
Turnaround Time = 41

Process 6
Finish Time = 78
Wait Time = 50
Turnaround Time = 62

Average Finish time = 48.5714
Average waiting time = 32
Average Turnaround time = 42
CPU Utilization = 88.6076%
```

Calculating different values:

- Wait and Finish times have been shown in each method
- Turnaround time = `finishTime[i] - process[i].arTime`
- Average finish Time :

```
float avgFinish = accumulate(finishTime.begin(), finishTime.end(), 0) / (finishTime.size() * 1.0);
```

- Average Wait Time :

```
float avgWait = accumulate(waitTime.begin(), waitTime.end(), 0) / (waitTime.size() * 1.0);
```

- Average Turnaround Time :

```
float avgTurn = accumulate(turnaroundTime.begin(), turnaroundTime.end(), 0) / (turnaroundTime.size() * 1.0);
```

- CPU utilization: `optimalCPUusage / (CPUusageOfMethod * 1.0)) * 100`
  Where optimal CPU usage is the cpu usage with no context switches
  And CPU usage of Method is the finish time of each method.

The Gantt Chart:

Thanks to Mr. Omar Aburish for letting me know that there was a library called (TextTable) for table representation in C++

How it draws the gantt chart:

In each method there are two rows and at each iteration the process ID and **if needed** the word (CS) is added. The second row consists of start and finish times for each process and the start and finish for Context Switch

```
for (int i = 0; i < processFCFS.size(); i++) //first row of gantt chart
    {
        tableInput = (to_string(processFCFS[i].processID));
        table.add("P" + tableInput + " ");
        if (processFCFS.size() - i != 1)
        {
            table.add("CS");
        }
    }
```

```
for (int i = 0; i < processFCFS.size(); i++) //second row of gantt
chart
   {
     tableInput = "";
     tableInput = to_string(progress);
     tableInput += (" " + to_string(progress));
     table.add(tableInput);
     if (processFCFS.size() - i != 1)
       {
      progress += CS;
      tableInput = (to_string(progress) + " " + to_string(progress +
CS));
      table.add(tableInput);
       }
   }
```

Output:

```
/\/\/\/\/\/\/\/\/\ FCFS /\/\/\/\/\/\/\/\/\
x----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x
|P1  |CS   |P2   |CS   |P0   |CS   |P3   |CS   |P4   |CS   |P5   |CS   |P6   |
x----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x
|0 12|13 14|13 16|17 18|17 27|28 29|28 49|50 51|50 57|58 59|58 63|64 65|64 76|
x----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x-----x
```

MultiThreading:

Using the C++ library <thread> we can simply assign each function to a thread as shown below:

```
   //MULTI THREADING
   thread t1(FCFS); //create first thread(FCFS)
   t1.join(); //start thread
   thread t2(SJF); //create second thread(SJF)
   t2.join(); //start thread
   thread t3(roundRobin); //create third thread(RR)
   t3.join(); //start thread
```

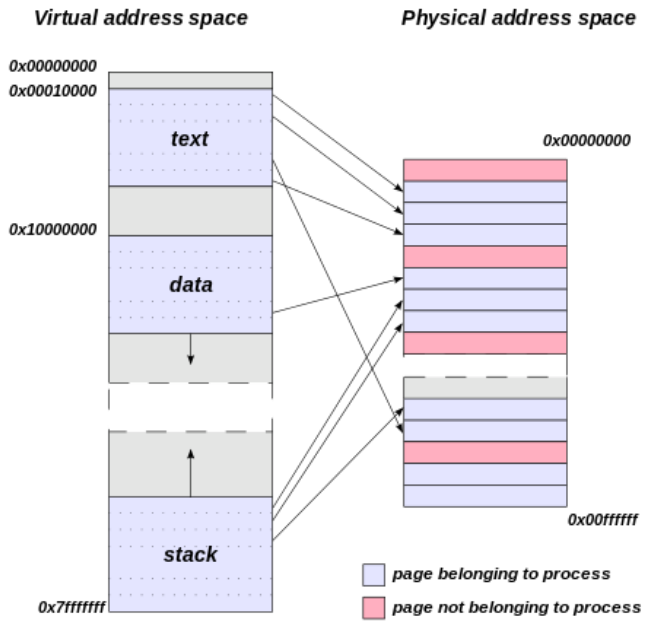
These functions will execute in parallel.

Phase 2:

Simulate how a computer memory works, showing how each process is given a select number of frames that no other process can access, and show how the memory looks after all processes are assigned their frames.

The page table:

A page table is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses.



Page table code implementation:

First the program calculates the number of pages available from memory size, and the number of pages it needs, if the needed are larger than available the program stops. After that an array is filled with the number of available pages and all marked free. Each process is split into a number of pages depending on it size

$$process\ pages\ = \frac{process\ size}{frame\ size}$$

After that using the rand() function with a random seed each time the program runs it takes a processe's page and generates a random frame from the available pool, if the frame is taken it repeats the process

```
for (int i = 0; i < numberOfProcesses; i++) // this checks if a spot in
memory is reserved, if not it takes it
    {
    pageTable.push_back(vector<pair<int, int>>());
    while (processPages[i] > 0)
      {
      int randomVal = rand() % (max_number + 1 - min_number) + min_number;
          if (memory[randomVal] == false)
           {
            pageTable[i].push_back(make_pair(processPages[i], randomVal));
            memory[randomVal] = true;
            processPages[i]--;
            }
        }
    }
```

After its assignment is complete the program moves onto the next page until all the process's pages are assigned a frame, then it moves to the next process.

Sample Output:

```
x-------------x--------------x
| Process 0   |              |
x-------------x--------------x
| page number | frame number |
x-------------x--------------x
|     16      |     2        |
x-------------x--------------x
|     15      |     86       |
x-------------x--------------x
|     14      |     53       |
x-------------x--------------x
|     13      |     49       |
x-------------x--------------x
|     12      |     95       |
x-------------x--------------x
|     11      |     35       |
x-------------x--------------x
|     10      |     66       |
x-------------x--------------x
|     9       |     68       |
x-------------x--------------x
|     8       |     109      |
x-------------x--------------x
|     7       |     84       |
x-------------x--------------x
|     6       |     46       |
x-------------x--------------x
|     5       |     96       |
x-------------x--------------x
|     4       |     56       |
x-------------x--------------x
|     3       |     85       |
x-------------x--------------x
|     2       |     65       |
x-------------x--------------x
|     1       |     33       |
x-------------x--------------x
```

Memory Map:

After the program assigns a frame to a page it marks that place as “Busy” in memory so no other process’s page can access that location, it work by changing the value in the memory array to “True” instead of the initial value of false.

```
if (memory[randomVal] == false)
    {
      pageTable[i].push_back(make_pair(processPages[i], randomVal));
      memory[randomVal] = true;
      processPages[i]--;
     }
```

The program can also show what percentage of the memory is busy (frame is assigned to a page)

```
Free space = 30.4688%
Busy space = 69.5312%
```

Sample Output:

```
Logical Memory Status:

x-------------x-------------x
| Frame number | Frame status |
x-------------x-------------x
|      0       |     Busy     |
x-------------x-------------x
|      1       |     Busy     |
x-------------x-------------x
|      2       |     Busy     |
x-------------x-------------x
|      3       |     Busy     |
x-------------x-------------x
|      4       |     Free     |
x-------------x-------------x
|      5       |     Busy     |
x-------------x-------------x
|      6       |     Busy     |
x-------------x-------------x
|      7       |     Busy     |
x-------------x-------------x
|      8       |     Free     |
x-------------x-------------x
|      9       |     Busy     |
x-------------x-------------x
|      10      |     Free     |
x-------------x-------------x
|      11      |     Busy     |
x-------------x-------------x
|      12      |     Busy     |
x-------------x-------------x
|      13      |     Busy     |
x-------------x-------------x
|      14      |     Busy     |
x-------------x-------------x
|      15      |     Busy     |
x-------------x-------------x
|      16      |     Free     |
x-------------x-------------x
```

Physical Address Calculation:

To calculate the physical address from the logical address we:

- Determine which frame is the address located at
- Determine the displacement of the address

The program calculates the address using this equation:

```
x = ceil(logicalAddress / (pageSize * 1.0));
y = logicalAddress % pageSize;
cout << "Physical Address is: " << (pageSize * x) + y << "\n";
```

Sample Output:

```
Enter logical address:
548
Physical Address is: 1060
```

Conclusion:

This program provides an overview of the different ways the CPU deals with processes and their scheduling, and it also demonstrates how all these processes are split into equal parts that are assigned to equal blocks in memory and how memory looks like after all the processes have been assigned to parts of it.

Notes:

- You can view the entire code at github: https://github.com/Samir-Abuisneneh/CPU-simulation

- For the TextTable Library visit: https://github.com/haarcuba/cpp-text-table

- When running the code make sure to have text with the processes and make sure to have downloaded the header file for TextTable

- To be able to use threads in c++, run the program using this command (this is on Linux and Windows10, maybe it's different of other operating systems):
  `g++ -std=c++11 -pthread main.cpp`