İSTANBUL TİCARET ÜNİVERSİTESİ

Operating Systems

Assist. Prof. Dr. Metin TURAN

Simulation of OS Tasks

Group Members:

Kaan Çembertaş - 200001684

Yiğitcan Güler – 200001709

Okan Arı – 2000001696

1. Introduction

1.1 Purpose

The purpose of this Project is that to implement the Operating System tasks such as Process Scheduling and Memory Management algorithms. The Project is designed and developed by a team of 3 members. Team members are Kaan Çembertaş, Okan Arı and Yiğitcan Güler.

1.2 Technology

The implementation of this simulation is written by Java programming language by using Object Oriented Programming principles.

2. Input

2.1 Input From User

The simulation asks to the user which Memory Placement and Process Scheduling algorithms will be used in the simulation when the simulation initializes.

2.1.1 Sample Input

```
---------------------
(1) - BEST FIT
(2) - FIRST_FIT
(3) - NEXT FIT
Please enter the Memory Placement Algorithm: 2
---------------------
(1) - FIRST IN FIRST OUT
(2) - SHORTEST JOB FIRST
(3) - ROUND ROBIN
Please enter the CPU Scheduling Algorithm: 1
```

2.2 Input File

Process Id	Size (KB)	Incoming Time (ms)	Execution Time (ms)	I/O Requests Time List	IO Burst Time
1	245	0	25	8,17	2
2	365	13	10	5	3
3	155	18	19	3,11,16	1

3. Flow of Simulation

The flow of simulation is described below step by step.

1. Read the input from the user and user selects the algorithms.
2. Read the processes from the input file (processes.txt) and create objects of the processes, save them to the ArrayList called processList.
3. Initialize the simulation
    a. Create a OS process object which process id is equals to 0 and size of 350 KB.
    b. Create a memory object which has size of 1024 KB, and place the OS into the memory.
    c. Create a CPU object by passing the time quantum which is 5 and passing the memory object.
    d. Start the simulation loop. The loop executes until when the Number of Finished Processes is equal to Number of Processes in the processList.
    e. When the simulation loop overs, write calculated datas to ouput files under the “output” folder.

4. Outputs

We will be have 3 output files under the “output” folder which are cpuOutput.txt, detailedCpuOutput.txt and memoryOutput.txt

4.1 Sample Cpu Output

In the cpuOut.txt file, we can see the execution of processes and IO bursts per millisecond.

```
[MS] [CPU] [IO]
0 P1 N
1 P2 P1
2 P2 N
3 P1 N
---------
---------
13 P4 P3
14 P5 P4
15 P6 P5
16 P7 P6
17 P3 P7
18 P3 N
----------
---------
30 P7 N
```

4.2 Sample Detalied Cpu Output

In the detaliedCpuOut.txt file, we can see lots of property of the simulation details. The **properties** are described below.

a. **Clock** of the CPU
b. **Free Areas** which holds the free areas in the memory and data structure of Linked List.
c. **Ready Queue** status which holds the Process Objects waiting for the cpu burst.
d. **IO Ready Queue** status which holds the Process Objects waiting for the IO burst.
e. **Current Process** which is executes on the CPU at that clock time.

f. **Current IO** which is executes on the **IO** at that clock time.
g. **Process Properties** which are in ready queue at that clock time. The Properties of Process Class is described in Process Class section **5.1.1**.

```
---------------------
Clock: 14
Free Areas: [350, 950]
Ready Queue: P6,P7,P3,P4,
IO Ready Queue: P5,
Current Process: P5
Current IO: P4

[PROCESS DETAILS IN THE READY QUEUE]
----------------------
```

In this one of the section of sample Detailed Cpu Output, we can understand the what goings on in the CPU, IO and Memory. For this example the current process at clock of 14 is P5, and in IO burst there is P4. In IO ready queue we can understand that P5 has IO burst after the clock 14 and when it is finished its process, it enters the IO Ready Queue and the process will be waiting for IO at next clock.

4.3 Sample Memory Output

In memoryOutput.txt file, we can see the status of the memory per milliseconds.

4.3.1 Sample Section of Memory Output

```
----- CLOCK: 1 -----

[P0] Process Id:Operating System
[P0] Begin Address: 0
[P0] End Address: 349

[P1] Process Id:P1
```

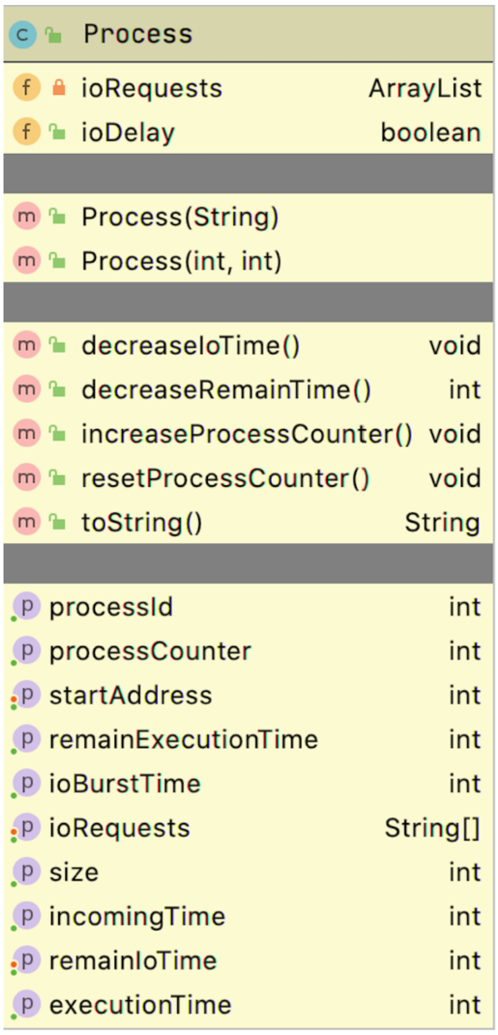
```
[P1] Begin Address: 350
[P1] End Address: 449

[P2] Process Id:P2
[P2] Begin Address: 450
[P2] End Address: 549
---------------------------
```

5. Classes

5.1 Process Class

5.1.1 Process UML



Process	
ioRequests	ArrayList
ioDelay	boolean
Process(String)	
Process(int, int)	
decreaseIoTime()	void
decreaseRemainTime()	int
increaseProcessCounter()	void
resetProcessCounter()	void
toString()	String
processId	int
processCounter	int
startAddress	int
remainExecutionTime	int
ioBurstTime	int
ioRequests	String[]
size	int
incomingTime	int
remainIoTime	int
executionTime	int

5.1.2 Process Properties

a. **Process Id:** The unique id of process.
b. **Process Counter:** The execution time that spends on CPU from the time the process last entered. Process Counter is an important property for preemption.
c. **Start Address** which is the address of the process placed in memory.
d. **Remain Execution Time:** The time in milliseconds that indicates Remain Time of CPU execution of the process.
e. **IO Burst Time** which is constant execution time of Process' IO burst.
f. **IO Requests** which is array of the integer and contains the remain IO bursts at that integer values of milliseconds.

g. **Process Size:** The size of area which occupied on memory.
h. **Incoming time (ms):** The time in milliseconds that when process trying to enter to the memory.
i. **IO Remain Time** which holds the remain IO burst time for a specific burst.
j. **Execution Time:** The total time that the process will spend on the CPU.
k. **IO Delay:** In the simulation loop we could have a case that when the Process exits from IO burst and take the CPU at the same CPU Clock. To prevent that case we use IO Delay property. When the process exits from the IO, we set the IO Delay true. When the IO Delay is true, the process cannot take the CPU, after a clock we set all the processes' IO Delay false. We can prevent that case by applying this method.

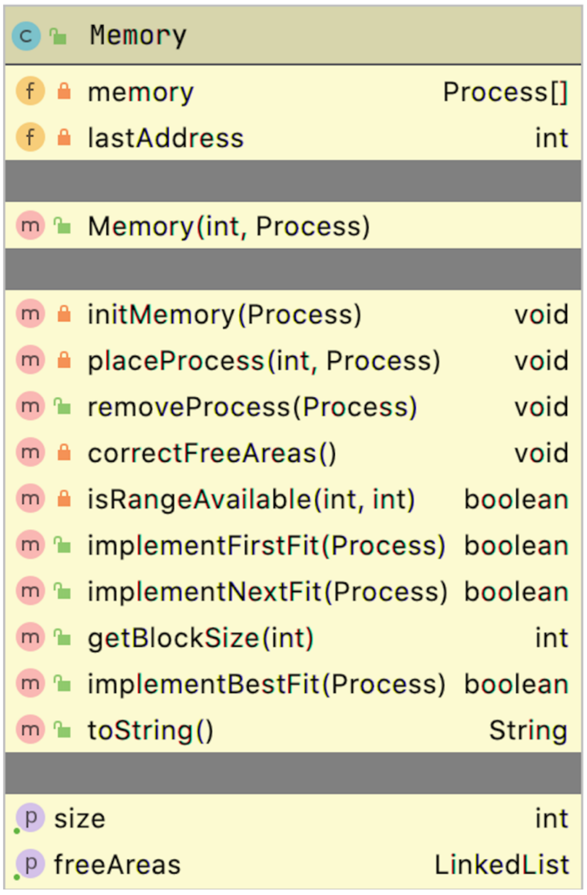
5.1.3 Process Methods

a. **Process(String):** The constructor of the process that initializes the process with input data which came from the input file.

b. **Process(int,int):** The constructor of the process that only initialized the OS Process with id of 0 and size of 350 KB.
c. **decreaseIoTime():** The function that decreases **IO Remain Time** when IO Burst applied.
d. **decreaseRemainTime():** The function that decreases **Remain Execution** Time when CPU Burst applied.
e. **increaseProcessCounter():** The function that increases **Process Counter** when CPU Burst appled.
f. **resetProcessCounter():** The function that resets the Process Counter. Process counter will be reset when the process enters to the IO Ready Queue or the process executed from Round Robin algorithm.
g. **toString():** The function that returns a String data which includes details of the process.

5.2 Memory Class

5.2.1 Memory UML



```
Memory
memory                        Process[]
lastAddress                   int

Memory(int, Process)

initMemory(Process)           void
placeProcess(int, Process)    void
removeProcess(Process)        void
correctFreeAreas()            void
isRangeAvailable(int, int)    boolean
implementFirstFit(Process)    boolean
implementNextFit(Process)     boolean
getBlockSize(int)             int
implementBestFit(Process)     boolean
toString()                    String

size                          int
freeAreas                     LinkedList
```

5.2.1 Memory Properties

a. **Memory:** An array which type of Process with size of memory size. The memory array provides us to simulate the Main Memory (RAM).
b. **Last Address:** An integer type of **lastAddress** variable provides us to save the last address which is searched from CPU with the placement algorithm of Next Fit.
c. **Size:** The size of memory which came from the Main Class.
d. **Free Areas:** The Linked List that holds the free areas in the memory in order.

5.2.3 Memory Methods

a. **Memory(int,Process):** The constructor of the Memory which has the parameters of memorySize and Operating System that type of Process.
b. **initMemory(Process):** The function to initialize the memory.
c. **placeProcess(int,Process):** An important method of Place Process provides us to place a process to the memory on start address. The start address and process are came via parameters.
d. **removeProcess(Process):** The function Remove Process provides to remove a specific process on the memory.
e. **correctFreeAreas():** The function that provides to correct the logical errors on the Free Areas Linked List.
f. **isRangeAvailable(int,int):** The function checks the memory if the bounds which came via parameters are available.
g. **getBlockSize(int):** Function that returns the block size of free area which starts from startAddress which came via parameter.
h. **implementFirstFit(Process):** The function that implements the First Fit algorithm for the process. The function searches on the memory with start addresses in the Free Areas, and by using isRangeAvailable method, the Process is placed on the first free area in the memory by this function.
i. **implementBestFit(Process):** The function that implements the Best Fit algorithm for the process. The function iterates all the

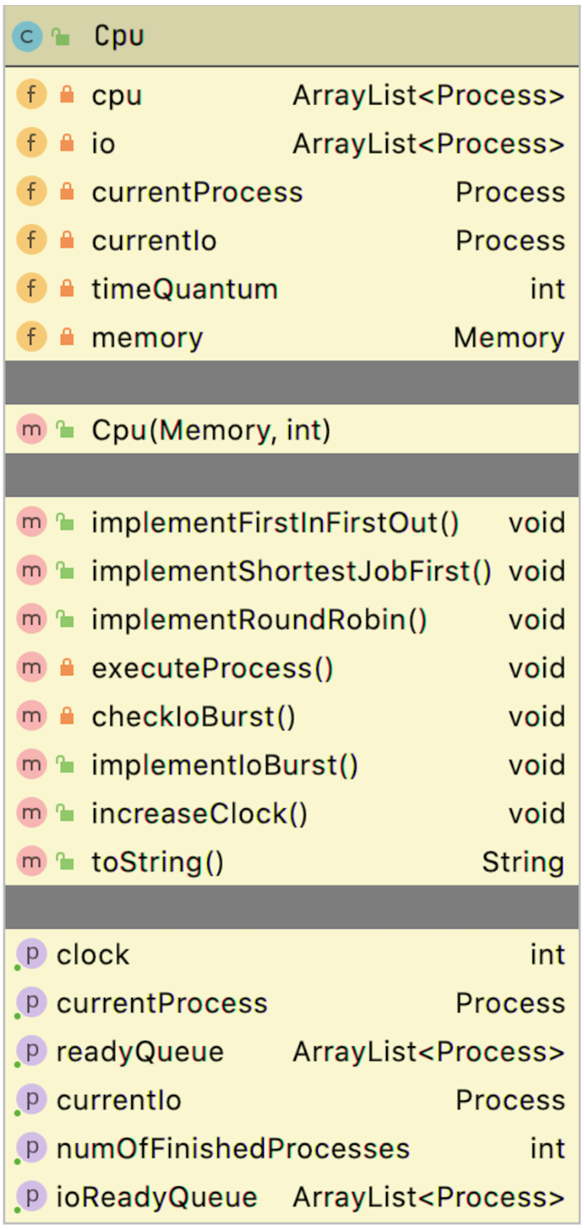
all the memory, and the Process placed on a free block which remains the smallest free area on memory.

j. **implementNextFit(Process):** The function that implements the Next Fit algorithm for the process. The function does the same thing with First Fit but the method start searching from the **Last Address** property. After placing the process, last address will be updated.

k. **toString():** The function that returns a String data which includes details of the memory.

5.3 CPU Class

5.3.1 CPU UML



```
Cpu

cpu                          ArrayList<Process>
io                           ArrayList<Process>
currentProcess               Process
currentIo                    Process
timeQuantum                  int
memory                       Memory

Cpu(Memory, int)

implementFirstInFirstOut()   void
implementShortestJobFirst()  void
implementRoundRobin()        void
executeProcess()             void
checkIoBurst()               void
implementIoBurst()           void
increaseClock()              void
toString()                   String

clock                        int
currentProcess               Process
readyQueue                   ArrayList<Process>
currentIo                    Process
numOfFinishedProcesses       int
ioReadyQueue                 ArrayList<Process>
```

5.3.2 CPU Properties

a. **Cpu:** The variable of CPU which type is ArrayList<Process> holds the Processes that executes on CPU per milliseconds.
b. **IO:** The variable of IO which type is ArrayList<Process> holds the Processes that executes on IO per milliseconds.
c. **Current Process:** The process type of variable indicates the process that executes on CPU at that clock.
d. **Current IO:** The process type of variable indicates the process that executes on IO at that clock.
e. **Time Quantum:** Integer type of variable indicates the time in milliseconds for preemption.
f. **Memory:** The memory object created for the simulation. We need this memory object in CPU class to apply some methods such as Remove Process.
g. **Clock:** Time in milliseconds that indicates the simulation time.
h. **Ready Queue:** Queue that contains processes which are waiting for CPU execution.
i. **IO Ready Queue:** Queue that contains processes which are waiting for IO execution.
j. **NumOfFinishedProcesses:** The variable holds the number of processes that are removed from memory.

5.3.3 CPU Methods

a. **CPU(Memory,int):** The constructor method of CPU Class. It initializes currentProcess and currentIO to null, and sets the timeQuantum and memory.
b. **executeProcess():** The method processes the current Process if it is exists. And after execution, the method removes process from the memory if the process' remain time is equal to 0.
c. **checkIoBurst():** The method checks Process if it has IO burst at the next clock after the CPU execution. If it has, current process enters to IO ready queue.
d. **implementIoBurst():** The method gets process from IO ready queue and it runs at IO.
e. **increaseClock():** The method increases the CPU clock time.

f. **implementFirstInFirstOut():** The method gets a process from ready queue if the current process is null and it runs until process finished.If process has IO request then it goes to IO.

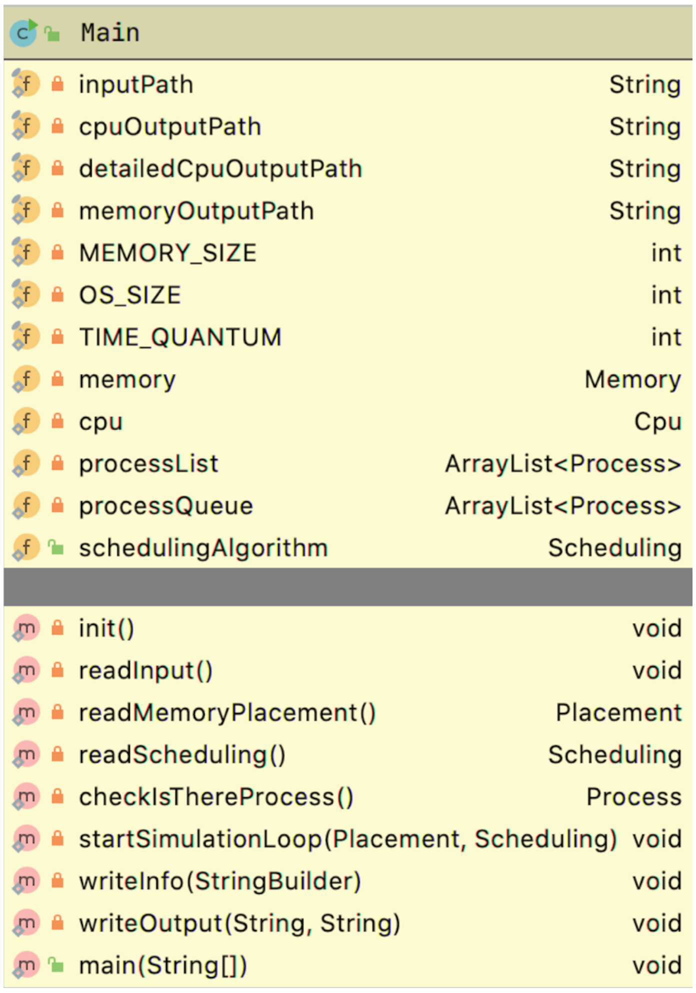
g. **implementShortestJobFirst():** If the current process is null then the method select a process which has the smallest remain time from the ready queue.

h. **implementRoundRobin():** First, if the current process is not null then the method checks the if current process' counter is equal to time quantum.If so the current process is thrown.After this check if the current process is null then the method gets a process from ready queue.

i. **toString():** The function that returns a String data which includes details of the CPU.

5.3 Main Class

5.3.1 Main UML



Main	
inputPath	String
cpuOutputPath	String
detailedCpuOutputPath	String
memoryOutputPath	String
MEMORY_SIZE	int
OS_SIZE	int
TIME_QUANTUM	int
memory	Memory
cpu	Cpu
processList	ArrayList<Process>
processQueue	ArrayList<Process>
schedulingAlgorithm	Scheduling
init()	void
readInput()	void
readMemoryPlacement()	Placement
readScheduling()	Scheduling
checkIsThereProcess()	Process
startSimulationLoop(Placement, Scheduling)	void
writeInfo(StringBuilder)	void
writeOutput(String, String)	void
main(String[])	void

5.3.2 Conclusion

The Main is the runner class of the simulation. The simulation loop, Input / Output control mechanisms, constant values and object references (Cpu,Memory,Process) are in the main class.

The most important method of this class is the StartSimulationLoop. After reading the inputs (Processes, Cpu Scheduling and Memory Management algorithms) the simulation starts with this method.

The simulation loop executes until the when number of finished processes is equal to number of all processes which is came from processes.txt input file.

The simulation checks if there is a process that came at that CPU clock. If so, the simulation tries to place the process with user selected algorithm. After that the simulation executes the IO burst, after that executes the CPU burst with user selected CPU Scheduling algorithm. These steps described above occurs in every milliseconds of simulation.

When the simulation overs, the datas calculated by the simulation will be written to the output files which locate in “outputs” folder in the source code of direction.