Ayoub Mabkhout

Hanane Nour Moussa

CSC 3351 Operating Systems

CPU Scheduler and Page Replacer in Java

Project Overview:

In order to illustrate the concepts that we have learned in CSC 3351 this semester, our team decided to build a **desktop application using Java** that would simulate the most vital functionalities of an OS, namely **CPU scheduling** (FCFS, Round Robin, SPN, SRT) and **page replacement** (FIFO, LFU, LRU, Clock). The program is composed of two main parts: **a random process generator** that simulates the end user, and the **operating system** which performs CPU scheduling and page replacement.

Specifically, this project targets the following ILOs:

- ✓ **Multithreading:** In our programs, we have made use of three different threads (ProcessGenerator, Process, and Scheduler).
- ✓ **Synchronization:** In order to synchronize the different threads, we have made use of Java synchronization capabilities, namely wait/notify.
- ✓ **CPU scheduling:** the first main OS functionality that our program emulates is CPU scheduling. We have implemented the 4 algorithms covered in class:
  - FCFS: The simplest scheduling policy that follows a first come first served or FIFO order. As each process becomes ready, it joins the ready queue. When the current process ceases execution, the longest waiting process in the ready queue is selected.
  - Round Robin: Similar to FCFS, Round Robin introduces preemption based on a clock. Each process is allowed to run for a defined allotted amount of time, called the time quantum, before the longest waiting process in the ready queue is chosen to run.
  - SPN: Non-preemptive policy in which the process with the shortest expected processing time is chosen to run next.
  - SRT: Preemptive version of SPN in which the scheduler chooses the process with the shortest remaining processing time.
- ✓ **Page replacement:** the second functionality we have implemented is page replacement. We have implemented the 4 algorithms covered in class:
  - FIFO: in which pages are replaced in a First In First Out manner.
  - LFU: in which the page with the smallest access frequency is replaced.
  - LRU: in which the page least recently accessed is replaced.
  - Clock: in which each page is associated with a used bit. If the used bit is 0, this page is replaced, otherwise, the use bit is set to 0 and the page is replaced in the second iteration if the used bit is not reset to 1 upon being accessed.

Code documentation:

Our code is partitioned into the following packages and classes:

- `end.user`:
  - `ProcessGenerator.java`: Generates processes with random start times, bursts, pages, and page sequence.
  - `SystemClock.java`: Manages the operations timing in terms of the CPU speed.
- `gui`:

- `OSFrame`: Generates the GUI, initializes the scheduler and memory manager, and updates the display
- `ProgramState`: Saves the state of the running processes and computes performance measurements

- `processes`:
  - `Frame.java`: represents a frame in main memory into which pages are loaded.
  - `MainMemory.java`: a collection of frames.
  - `Page.java`: represents a process frame.
  - `PageTable.java`: aggregation of pages. Each process has a page table.
  - `Process.java`: represents a process.
  - `SecondaryStorage.java`: a collection of processes stored in Secondary Storage.
- `system.managers`:
  - `Scheduler.java`: superclass of all scheduling algorithms.
  - `FCFS.java`: implementation of FCFS scheduling algorithm.
  - `RR.java`: implementation of Round Robin scheduling algorithm
  - `SPN.java`: implementation of Shortest Process Next scheduling algorithm
  - `SRT.java`: implementation of Shortest Remaining Time scheduling algorithm
  - `MemoryManager.java`: superclass of all page replacement algorithms.
  - `FIFO.java`: implementation of First In First Out page replacement algorithm.
  - `LFU.java`: implementation of Least Frequently Used page replacement algorithm.
  - `LRU.java`: implementation of Least Recently Used page replacement algorithm.
  - `Clock.java`: implementation of Clock/Second Chance page replacement algorithm.
- `system.managers.comparators`:
  - `LFUComparator.java`: definition of comparator used for LFU priority queue.
  - `LRUComparator.java`: definition of comparator used for LRU priority queue
  - `SRTComparator.java`: definition of comparator used for SPN/SRT priority queue (SRT is just the preemptive version of SPN).
- `gui`:
  - `OSFrame.java`: implementation of the graphical user interface.

Class description:

Table 1: Summary of Classes in the Program

Class	Attributes	Methods	Superclass extended	Interface implemented

`OSFrame`	`completedProcesses`
	`run //enum running`
	`timer`
	`refreshPeriod`
	`startTime`
	`endTime`
	`sch// scheduler scheme, enum`
	`mem// MM scheme, enum`
	`processQueue`
	`userThread`
	`systemThread`
		`Event Handlers`
		`main()`
			`JFrame`	_____

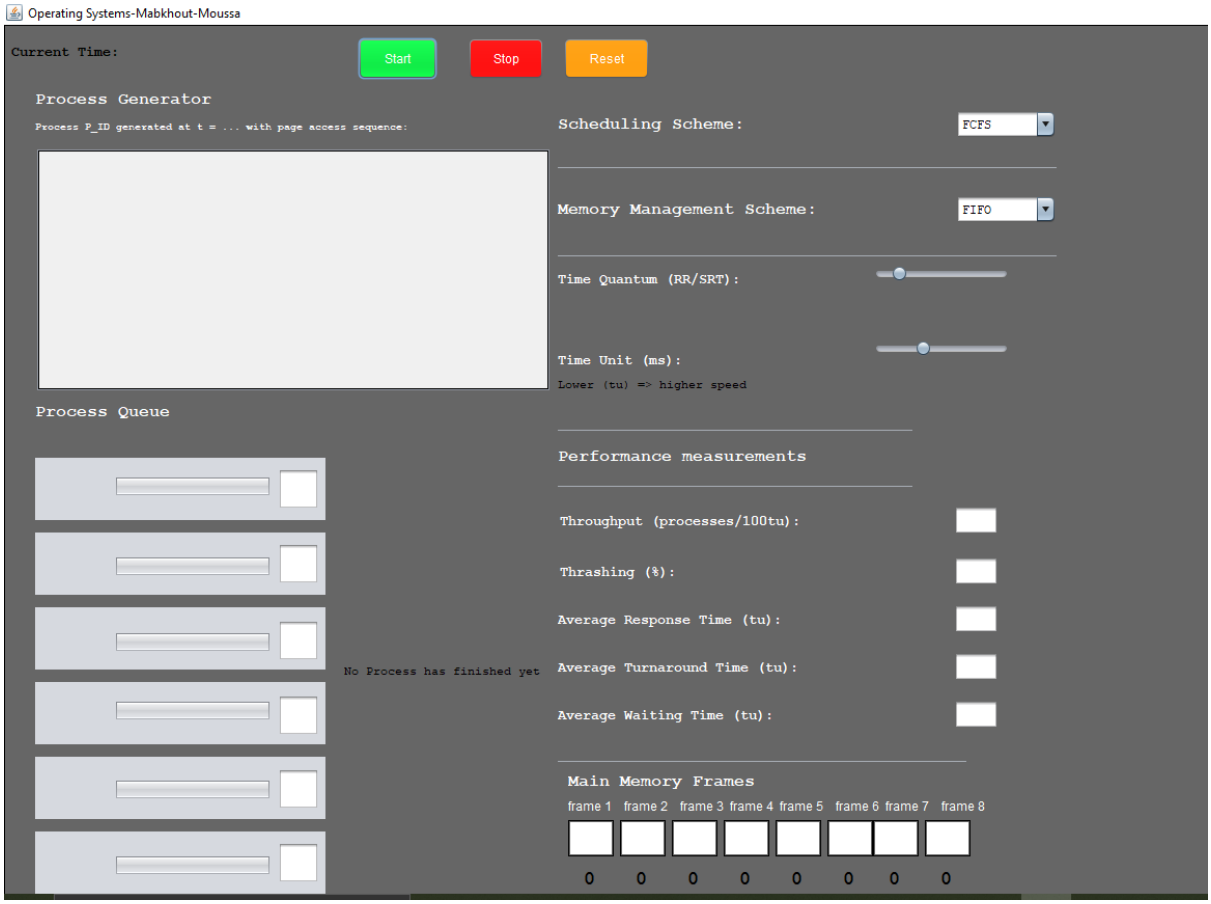
	guiUpdater			
Updater	processList processNames processProgressBars workingPages frames usedBits	updateProcessList() updateProcessDisplay() updateFrameDisplay() resetFrameDisplay() reserProcessDisplay()	_____	_____
GUIUpdater		run()	_____	Runnable
ProgramState	processQueue responseTimes turnaroundTimes waitingTimes	newestProcess() replacedProcess() lastEnqueuedProcess() getResponseTimes() getTurnaroundTimes() getWaitingTimes() getThrouput() getThrashing()	_____	_____
ProcessGenerator	processQueue queuesize avgSubprocessLength variation avgPageNumber avgPageSize	run() generate() generateProcess() loadProcesses()	_____	Runnable
SystemClock	timeQuantum bootTime	waitFor() getTime()	_____	_____
Frame	frame_ID size full	_____	_____	_____
MainMemory	totalMemory frames	init() getNextFreeFrame() isFull() showFrames()	_____	_____
Page	frame size loaded lastUsed frequency used p_ID	load() unload()	_____	_____
PageTable	pages size	addPage()	_____	_____
Process	p_ID status arrivalTime startTime burstTime remainingTime pageTable sequence	resumeSubprocessTime() runSubprocess() runSequence() isDone() getCurrentPage() run()	_____	Runnable
SecondaryStorage	processes	_____	_____	_____
Scheduler	processQueue size	enqueueProcess() dequeueProcess() runNext() run()	_____	Runnable
FCFS	_____	@Override runNext()	Scheduler	_____
RR	_____	@Override runNext()	Scheduler	_____
SPN	_____	@Override	Scheduler	_____

		runNext()		
SRT	_____	@Override runNext()	Scheduler	_____
MemoryManager	loadRate loadedPages	loadPage() load() replace() enqueue()	_____	_____
FIFO	_____	@Override replace() @Override dequeue()	MemoryManager	_____
LFU	_____	@Override replace() @Override dequeue()	MemoryManager	_____
LRU	_____	@Override replace() @Override dequeue()	MemoryManager	_____
Clock	_____	@Override replace() @Override dequeue()	MemoryManager	_____
LFUComparator	_____	@Override compare()	_____	Comparator<Page>
LRUComparator	_____	@Override compare()	_____	Comparator<Page>
SRTComparator	_____	@Override compare()	_____	Comparator<Process>

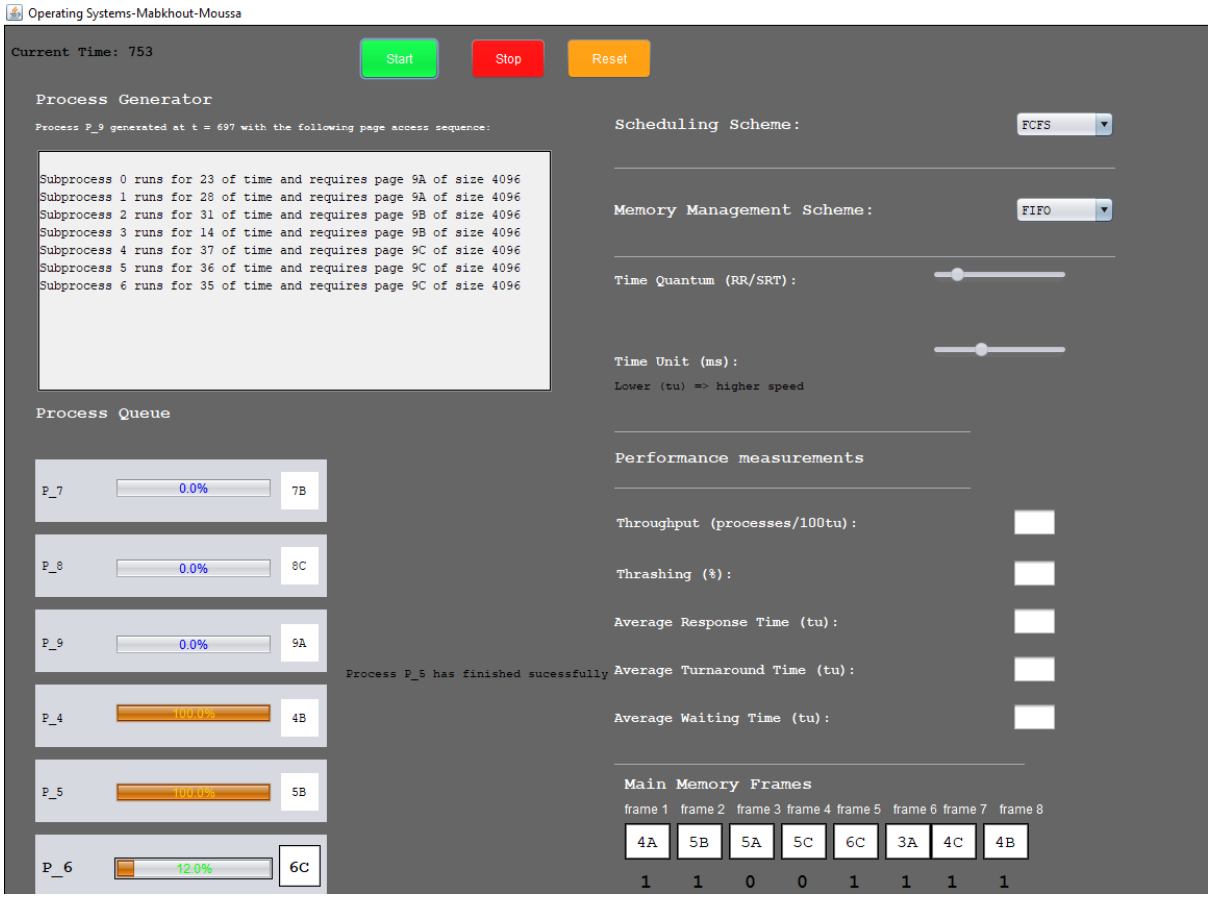
Note: the absence of methods means that they only consist of getters and setters.

Note: the absence of attributes means that they are inherited from a superclass.

Screenshots and user guide:



Our app's user interface



Our app while running

Once the app is run, we can **select the scheduling scheme and memory management scheme** by changing the values in the comboboxes. Using the slider, the time quantum and time unit can also be changed.

Once we click on the start button, the text area starts showing the **random processes** being generated. The **pages** belonging to the processed, as well as the sequence in which they will accessed in shown.

The processes then start executing according to the selected scheduling scheme. The currently **executing process** is shown in bold (P_6 in the screenshot above). The progress bar shows how close the process is to completion. **The page that the process is currently using** is shown in the square label right next to the progress bar.

The main memory (right lower corner) shows **the frames** and the pages they contain. The **used bit** of each page is shown underneath it (used in the Clock page replacement algorithm).

Once we click on stop, the **performance measurements** are computed and shown.

A **demo** of the project can be found via the following link: https://alakhawayn365.sharepoint.com/sites/me8/Shared%20Documents/Forms/AllItems.aspx?id=%2Fsites%2Fme8%2FShared+Documents%2FGeneral%2FRecordings%2FProject+Demo-20220511_015816-Meeting+Recording.mp4&parent=%2Fsites%2Fme8%2FShared+Documents%2FGeneral%2FRecordings&isSPOFile=1

Discussion:

The classes shown in Table 1 interact as follows:

- The **random process generator** and the **process scheduler** interact according to a **"producer-consumer"** manner. The shared buffer in this case is the process queue. Its maximum size is 6. Both the process generator and the scheduler implement the **Runnable** interface and are **synchronized** using **Java's wait/notify mechanism**.
- The Scheduler is extended by four classes: **FCFS, RR, SPN, and SRT**. Each one of these classes implements the logic of its respective algorithm.
- Once the scheduler is running, it makes a call to the **Memory Manager**. The Memory Manager then takes in the state of the running process and loads its **pages** from **secondary storage** to **main memory**.
- The Memory Manager is extended by four classes: **FIFO, LFU, LRU, and Clock**. Each one of these classes implements the logic of its respective algorithm.
- The OS Frame contains the code for the graphical user interface as well the driver main() method.
- The GUI is **updated periodically** to show the current state of the running processes. To this end, we have decided to define a class called GUI Updater. This class implements **Runnable** and is used from the Updater class to update the display according to the refresh period.
- The choice to implement the GUI Updater as a separate thread comes as an alternative to implementing a change listener, for which the implementation would have been taxing since the GUI would have to change whenever the smallest change occurs. Although the thread implementation leads to some delays and lags, it remains the most straightforward one from an implementation point of view.

The first phase of the project consisted of implementing the logic of random process generation, scheduling, and page replacement. This was fairly straightforward since it mostly consisted of mapping our code to various OS concepts we have covered in class.

The second phase, which was the most challenging and time consuming one, consists of linking the GUI to the program logic, all while ensuring a friendly user experience (minimal lagging, descriptive UI, etc.). Of course, there is a lot of improvement that could have been done at this level, but the current state of our app is usable.

Conclusion:

This project was a valuable opportunity to both implement the concepts that we previously learned conceptually in class as well as to learn new multithreaded programming skills using the Java language.

The project could either be run through an IDE (preferably NetBeans) or through the following path:

OS Project ➔ Operating System ➔ dist ➔ Operating_System.jar

Enjoy your mini-OS!