

## Project Description

### Operating system simulator

#### 1. Summary

This project aims at designing and implementing a simulator of an operating system. Students are expected to prepare a computer program that will serve as a simulator of creation and running of processes on a theoretical operating system. These processes will originate from program files (also referred to as templates) provided by you.

#### Program files will be consisting of:

Pre-determined instructions that will simulate the usage of computer system resources. **These programs need not to be coded, only simulated.** These program files will have simulated instruction and I/O cycles that is proportional to the type and size of the program (e.g., template for printer driver will focus mainly on I/O cycles, while template for a spreadsheet will focus mainly on CPU cycles). **Simulator must create a requested number of processes from each of templates randomly using a pre-defined generator.** Classes are to be created to mimic the role of different hardware components and software embedded in the operating system. These classes will have methods in them to perform operating system duties and will rely or call on other classes or methods, thus creating a fully interconnected system.

**Note:** We are **NOT** creating an Operating System, **just Creating an Operating System Simulator.** We should be able to by the end Run a simulator of OS that is going to work on multiple programs. (this need the understanding of the inner workings of Operating Systems and the ability code all algorithms that are used to manage, overview, and account for processes happening in an operating System.

#### You will need to write your own:

- Scheduler,
- Process synchronization
- Memory management
- Memory cleanup scheme

And all will be done **using these Mock programs with random cycles**, which are explained in more detail below.

**Applications can be written in any language.**

You will be multithreading so **pick a language that has multithreading**, AND as long as you have a library that allows you to *create software threads* you will be good.

You will create **four components** in simulator. Which are listed below

**NOT ALLOWED** To use library with pre-coded schedulers, or pre-coded inter-process communication THESE must be written by YOU. YOU CAN use for memory manipulation, and libraries for opening files.

## 2. Project Details

This project will consist of four major components –

- (1) process management
- (2) memory management
- (3) I/O management
- (4) user interface.

**Each individual component is explained in detail below.**



### 2.1 Process management

“Program files”(also referred to as “templates”) **are text or .xml files that are** used to create processes that will be then managed and run by our OS simulator. Therefore, **“program file” must be treated as a template** that can be used to create numerous processes with diverse parameters (e.g. by randomization of the values in the template).

Note: that means you will have more than one template. I’m sure that you know about CPU bound and I/O bound processes. So, template for CPU bound process would consist of mainly CALCULATE commands, while I/O commands. The processes can be longer and shorter so templates can have few instructions (for short processes) or double-digits number of instructions (for long processes). Combine those together you have recipes for 4 templates.

Each template (and thus each process created from it) will **consist of a “mock” code** written in a form of keyword operations, each representing the usage of a different resource managed by OS.

For simplicity, **only three types of operations are required:**

- CALCULATE – When this line is read in, the simulator will run the process in the run state for the number of cycles specified as a parameter (i.e., occupy CPU for a given number of cycles , simulating the usage of CPU resource).
- I/O – This will put the process in the waiting state for a specified number of cycles.

- FORK – This will create a child process according to a selected parent-child management scheme. [Note: FORK will be used in phase 2](#)

[Note: Each “program file” must be created using different combinations of these commands \(they can appear multiple times\). Each command must have assigned number of cycles necessary for completion, as well as memory necessary for storing this command \(in case of a paging model being implemented\) .](#)

Example of a program file/template (this needs to be enriched with memory management in a later step):

Operation list:	min cycles	max cycles
CALCULATE	5	100
CALCULATE	25	50
I/O	10	20
CALCULATE	5	20
I/O	15	25

This template must be used as an input (seed) for a generator that will spawn a user-defined number of processes with randomized cycle length for each operation. Example of two processes created from the above template are:

Process #1		Process #2	
CALCULATE	46	CALCULATE	82
CALCULATE	33	CALCULATE	48
I/O	17	I/O	11
CALCULATE	11	CALCULATE	19
I/O	22	I/O	16

**Each program must contain at least one critical section and at least a single critical section resolving scheme must be implemented.** Critical sections can be implemented as tags encoding a number of operations that are considered as critical section. Critical sections can be hard coded into a template or placed randomly in each created process. **All information about each process must be stored in Process Control Block (PCB).**

**Note: Knowing how PCB is structured. We won't use all of the options in the simulator so you can choose the ones that are necessary for your specific design**

Single level child-parent relationship must be implemented within each template. It may be either deterministic (a child process is always created) or probabilistic (a child process has a fixed chance to be spawned).

**At least a single inter-process communication scheme, a message passing scheme, and a multi-level child parent relationship to be implemented.**

**NOTE:** In conclusion, we will need to create “program files” which are templates for a generator that will use our “program files” template and create different “types” of processes to be simulated via our project.

- By “types” mean that each “program file” we make is going to be a template used to generate a process that is supposed to represent an action that would be done on a computer by a user and handled by the OS. Such as printing a document or doing calculations in an Excel Sheet.
- The “program file” is going to simulate a complex program via three operations
  - CALCULATE
  - I/O
  - FORK
- Should be constructed with ranges for CPU cycle values for the above operations so that the generator can create possibly thousands of unique processes to be simulated
- So our “Program Files” are used to generate representations of actions on the computer.
- Template is input for any random number generator that we will define, that will allow us to create infinite number of processes with the same structure but with randomized cycle values.
  - WHY,
  - Because the expectations are to have hundreds if not thousands, processes running at same time. IF you had to input every process manual it would SUCK, so

- - DESIGN SIMULATOR so that it could generate up to even 100000 processes from template.
- Create specific templates in phase 1 with some of them being longer, shorter, some with 30 operations, some with maybe only 7. We may create them randomly, with either creating a random order of operations or length. Up to you!
- WE WILL NEED a set of at least 5 diverse templates used to generate user specified number or processes, each process with randomized cycle values.

## 2.2 Process lifecycle

The process created from templates must be managed by the OS simulator following a process lifecycle that is realized as switching among the possible states in which each individual process will be in a given moment:

- NEW – The program or process is being created or loaded (but not yet in memory).
- READY – The program is loaded into memory and is waiting to run on the CPU.
- RUN – Instructions are being executed (or simulated).
- WAIT – The program is waiting for some event to occur (such as an I/O completion).
- EXIT – The program has finished execution on the CPU (all instructions and I/O complete), releases resources and leaves memory.

## 2.3. Process scheduling

You must choose and implement a scheduling algorithm. Easiest solution would be to go with Round Robin approach due to the ease of implementation, however other solutions (with exception of a trivial First Come First Served) also will be accepted.

**It is required to implement at least two different schedulers must and compare them with regard to their performance, as well as assign priorities to processes, implement a multi-level queue scheduling, process resources, and deadlock avoidance algorithm.**

NOTE: So Two different schedulers, and the comparison of their performance as well as priorities to processes, implement a multi-level queue scheduling, process resources, and deadlock avoidance algorithm ARE PHASE 3 REQUIREMENTS! However, you can get it done anytime you want

## MEMORY MANAGEMENT NOT REQUIRED FOR PHASE 1

### 2.4. Memory management

Memory Management will be implemented by keeping a running total of all processes in main memory by not letting the overall memory size exceed the set limit. **The OS simulator will have a main memory size of 1024MB.** This value which is to be compared against the memory requirements of newly arrived processes. If total memory minus used memory is more than the newly arrived job's memory requirement, it may enter the READY state (queue). Otherwise, the process would remain in the NEW queue (if it has been just spawned) or in the WAITING queue (if it tries to re-enter the ready state).

**It is required for memory to be organized into a hierarchy consisting of three levels: main memory, storage, and registers (with all the necessary migration between these hierarchy levels), as well as to implement virtual memory and paging.**

### 2.5. I/O management

Two types of I/O events must be **handled**:

- coming from a process (i.e., generated by I/O command in the process) – as described in the section 2.1. Process management.
- **caused by external events (e.g., hardware peripherals) that may happen at any moment and caused by processes being executed by CPU.** This should be realized as a random event with a given probability of occurring during every cycle.

### 2.6. Multi-threading and multi-CPU

Multi-threading must implement using software libraries and take advantage of hardware threads in the processor. Single-thread based simulation is not accepted.

Additionally, for points necessary for B or A grades a simulation of multi-core and multi-thread architecture with at least two cores and four threads per each must be implemented. This can be achieved by dividing existing hardware threads into simulates

CPU classes and assigning separate schedules to each CPU instance.

### 2.7. User interface

This program/simulator will incorporate a user interface so that she/he can control the flow of the operating system and observe the “running” of it for testing purposes. It must be possible to load a program or job file automatically/manually into the simulator thus to conduct the

allocation of the program's PCB and memory space. The user can also specify the number of cycles to run before pausing. Statistics of the simulator should be available upon request, such as number of processes in each state, journal log of the simulator, and what resources are currently being used.

Additionally, for points necessary for B or A grades it is required to implement a Graphical User Interface (GUI). The GUI will display real-time statistics, visualizations, and data on all currently running processes. The PCB information of the jobs that are in memory and in the RUN state will be shown in some sort of GUI data table. This GUI table is always "refreshing" or updating as the simulator runs on its own or steps through the code.

### 3. General rules for the project

- This is an **open-ended project** and the design skills, capability of adapting to specifications where necessary, as well as imagination and creativity of solving the required tasks play a crucial role.
- Project must be fully functional, and the code of project must compile and run.
- Project must be a compound system of modules working together, not a collection of non-integrated parts.
- Source code must be delivered, as well as an executable version

of the project.

- Obligation to deliver a .pdf documentation of the project, discussing the implemented solutions, software, and hardware requirements for the project to run, as well as guidelines on how to run the project.



#### 4. Project requirements **I HAVE REORDERED THIS INTO PHASE ORDER**

Required functionality	Points	Phase of the Project
Process implementation and PCB, loading external processes and generating new ones on user request	5	Phase 1*
Critical section within each process and critical section revolving scheme	4	Phase 1*
Scheduler	5	Phase 1*
UI	1	Phase 1*
Single level parent-child relationship	5	Phase 2**
Basic Memory and operations on it	5	Phase 2**
I/O interrupts and handlers	3	Phase 2**
Multi-threading	7	Phase 2**
Multi-threading and multi-core via simulation	2	Phase 3***
Two inter-process communication methods	2	Phase 3***
Multi-level parent-child relationship	2	Phase 3***
Two schedulers and comparison	2	Phase 3***
Process priorities	1	Phase 3***
Multi-level queue scheduling	2	Phase 3***
Process resources	2	Phase 3***
Deadlock Avoidance Algorithm	2	Phase 3***
Memory divided into hierarchy	2	Phase 3***
Paging	2	Phase 3***
Virtual memory with paging	2	Phase 3***
Single inter-process communication method	2	Phase 3***
GUI with real-time monitoring and visualizations of simulator performance / operation	2	Phase 3***

## 5. Suggested modular approach to project organization and coding

