CS401: ADVANCED OPERATING SYSTEMS

Assignment 1

Due Date: Monday, 6 October 2025

In this assignment, you will implement the process management aspects of an Operating System. There are three main modules for implementation:

1. A set of data structures used by a typical operating system for managing processes and memory;
2. A *process* module that deals with various aspects of a process life-cycle; and,
3. A *scheduler wrapper* into which any scheduling algorithm may be plugged-in.

The OS simulator has a main function that has two steps: the first, calls a function or has a loop to create a number of processes; and the second, invokes a scheduler. The simulator is currently a single threaded program. It terminates when the scheduler returns. The scheduler returns when there are no processes in the system.

To help you with the design a header file, `ScisSos.h` is provided to you for inspiration. It can be changed by you to suit your design and ideas. It can also be used as is and is tested for working with a process manager and a scheduler written by me.

1 Data Structures

Implement the following data structures.

Process Instruction: This models a single executable statement in a process. It must have a `system call` and a `memory address reference`. The memory address reference is only a placeholder in this assignment. It will however need to be there for a later stage. There can be *just two* system calls: `short` and `long` (see Section **??**).

Process Control Block: A process control block (PCB) is created at the time of creating a process. It is a data structure that stores essential information for running a process as the process goes from one state to another. At a minimum, it must contain the following information:

pid (process ID), uid (user ID), size (number of executable statements), priority, process state, program counter, process type and memory behaviour (see below), pointer to executable code, page table and time slice.

Process: A process consists of three parts. The first part contains metadata such as the PID, Owner UID and its size (number of executable instructions). The second is the process control block. The third is the set of executable instructions.

Process Table: Process table contains pointers to the PCBs of each process such that
`process_table[pid] =` *`pointer to pcb of process having pid`*

Ready Queue: A list of PIDs of the processes in `ready` state.

Blocked Queue: A list of PIDs of the processes in `blocked` state.

Page Table: An array of page table entries <`page no.`>, <`frame no.`>, <`pid`>. Frame number can be empty if the page is not present in main memory.

Process Table: An array of PCBs one for each process in the system. The PID^{th} entry in the table is a pointer to the PCB of the process whose pid is *PID*.

2 Process

A process has three parts. The first is *metadata* such as its `PID, UID, size`, etc. Size is the number of executable instructions (see below) that make up the process. The second part is its PCB. The third is a sequence of executable instructions. Each instruction is a tuple $(< syscall >, < addressreference >)$. You can fill any value for $< addressreference >$ as we will not use it in this assignment but will be needed later when doing memory management.

The `syscall` entry is binary: '0' indicates that it is a `short` system call; '1' indicates that it is a `long` system call.

Create a `process` structure to represent a process.

3 PCB

Process control block is a structure that stores all the information necessary to represent a process and used for running it. The information is already given above. Create a PCB structure.

4 Scheduler

A scheduler is the main component of this assignment. A scheduler does the following steps:

1. Examine the entire process table and create or update a `ready queue` which contains the PIDs of the processes that are in `READY` state. Also, store the PID of the process that is currently in `RUN` state.

2. Call a scheduling algorithm with `ready queue` as input. The algorithm outputs a single PID. This is the process that will now be scheduled for running.

3. Update the PCB of the process which is currently in `RUN` state and change its state to `READY`. Get the PCB of the process with the PID returned by your scheduling algorithm and change its process state to `RUN`.

4. Call the process with the PID returned by the scheuling algorithm to run and terminate the scheduler.

Remember that the scheduler comes to life again when a running process calls it.

5 Running a Process

A process when it is scheduled as described in the previous section does the following. This is the second important part of this assignment.

1. Process starts from the instruction number given in the `program counter` of its PCB.
2. It continues reading its instructions (this is a simulation for executing those instructions) until one of the following conditions occur.
    - the current instruction contains a `long` system call
    - the number of instructions executed is greater than or equal to the *time-slice*.
3. The process stops reading its instructions and updates the PCB (as necessary).
4. The process calls the scheduler and terminates.

Remember that the process may be called again by the scheduler at some future time when it will resume execution from the program counter in its PCB.

6 Simulator Program

Simulator is the `main()` program of this assignment. It is quite simple. The first step is to create processes. You may want to put a loop and create around 10 processes with different sizes, priorities and process types.

The second step is to call the scheduler. Scheduler and processes interleave their executions until all the processes complete their execution. At that point the scheduler checks that no processes are in the `ready` queue and terminates itself.

7 Implementation Instructions

Please organise your code in multiple files if you implement the assignment in C language (preferred). You may also choose to implement it in C++, Java or Python although the last two are not really nice!

Here are some suggestions:

- Create a *header* file to store all your constants, data types and function declarations.
- Create one file for all process related activities such as `create process, run process, delete process, create PCB, print PCB`, etc.
- Create a separate file containing functions for the simulator. You really need two functions here: `initialise OS parameters` and `scheduler`.
- Create a separate file for the scheduling algorithm your group (see below) chooses to implement.
- Write a fourth file that contains the `main()` function.
- Compile the first three files separately (using the `-c` option to `gcc`) and then link them together when compiling the file containing the `main()` function.

You may form groups of two each for writing the overall simulator. Also, divide scheduling algorithms among the groups so that every group writes one scheduling algorithm. When it comes to submission, make sure you run your simulator with at least three scheduling algorithms and see how the processes are scheduled and run.

You will have to demonstrate the simulator to me or the TAs at a scheduled date and time after the due date which is **6:00 PM on Monday, 6 October 2025.**