CS2506 Operating Systems II Lab 1 31/01, 2/02/2017
Processes and Scheduling

Laboratory Goals / Objectives

Students will investigate the time-sharing round-robin scheduling of processes by designing and implementing the algorithm in pseudo-code and Python respectively. By the end of the lab, an understanding of the scheduling algorithm should be demonstrated, by providing a working simulation of the algorithm implementation.

Introduction

What is a process?

Processes are fundamental components of computing systems. They are instances of running programs.
In the Linux operating system, one root process, *init*, begins executing when the operating system starts. All other processes are created from within this process. Processes can therefore be seen as in a *parent-child* relationship. A parent process can create child processes. Init is an example of a *parent* process. All system processes that start from within the init process are *child* processes of init. Several system calls are provided to create processes, such as fork(). If process A calls fork(), a child process, process B, is created. A is the parent of B, and B is A's child. Process B will be an exact and separate duplicate/copy process of A, therefore B has the same executable code and data as A. B will start executing its code immediately after the fork() statement. Other calls for creating child processes are vfork(), clone(), and exec(). There are even different variations of exec().

How do processes work with the CPU?

Processes are all competing for the CPU(s). In a multiprogramming system, several processes appear to execute at the same time. However, if a computer system only has one CPU, only one process can execute at a time. Different processes are allocated different *time slices*, also known as *quanta* (pl.), which allow a process to take possession of the CPU for a certain limited period of time. A process that has taken possession of the CPU is said to be in the *running* state. In a single CPU system, only one process can be in the running state at a given time. If the process is not finished after the time slice has expired, its execution is pre-empted from the CPU and it goes back to the *ready state*, awaiting its next turn in the CPU. Another process takes control of the CPU. There are other ways a process can be removed from the CPU, even before its time slice has expired. If a process starts some I/O operation, such as waiting for keyboard input from the user, its execution is pre-empted from the CPU, but rather than return to the ready state, it enters the *blocked state* until the I/O operation it is waiting for has completed. When this occurs, it goes back to the ready state. Processes in the ready state are ordered in a queue "ready queue". Another possibility is the occurrence of an

interrupt that determines the suspension of the running program. After the interrupt is served, by running an interrupt handler, the suspended process is resumed.

What is the scheduler?

The *scheduler* is that component of the kernel that decides which process will take control of the CPU for the next quantum. How does the scheduler decide? There are various algorithms that can be used by the scheduler with different data structures of processes in the ready state.

Lab Work

In this lab we will explore the time-sharing round-robin algorithm. This algorithm always allocates the CPU for one quantum to the first process in the ready queue. We consider the following assumptions:

- All processes that were created and are ready to run are inserted in the ready queue on their arrival time (when their state becomes "ready") – they are ordered by the time of arrival.
- Each process needs a determined number of CPU time slices to complete (an integer, 1, 2, 3,...) which we assume to know.
- We'll consider that some processes will execute I/O operations. When such an operation is started, the process is pre-empted and switched to the blocked state, and the next ready process takes control of the CPU. The event of I/O completion triggers the switch of the process from the blocked state to the ready state – the process is inserted into the ready queue.
- We'll also consider that during the execution of a process, an interrupt can occur. If that happens, the process is suspended, the interrupt handler is executed, and then the process is restored and resumed.
- If the process completes its execution during a time slice, it will be terminated (exits the system), and the next process in the queue takes control of the CPU.

Tasks to Do

1. **Analysis**. First, draw the graph of process states and their links – the process lifecycle. Separately, draw the ready queue, the blocked queue, the CPU and place the scheduler between them. Consider all events that can occur with processes and the operations triggered by these events. For example, the arrival of a new process triggers its insertion in the queue, and an I/O completion triggers the switch of the process from the blocked state to the ready state. An interrupt triggers the suspension of the current process. What is the difference between the blocked and suspended transitions of a process?
   Then, consider all the actions associated with those events. Create a table with three columns, one for events and the second for the associated actions. For example, arrival (of a ready process) triggers →insert it into the ready queue. The third column is for comments.

2. **Design**. Outline the scheduling algorithm in pseudo-code. This will be your design for the algorithm. You should consider all aspects related to processes and scheduling. Your design of the algorithm must also take into account the possibility of having to block for I/O, in which case the process will forego the rest of its time slice, or to interrupt.

3. **Programming**. Provide an implementation of your algorithm in Python. You should design this with good object oriented design in mind, encapsulating the different parts of your solution into appropriate classes. For example, you may have one class to represent a process and its attributes such as the duration of execution expressed in terms of time slices, the presence of an I/O operation and its duration. You may use other classes to represent the ready and blocked data structures and their properties. Randomly, interrupts can occur.

   These classes may also contain data structures to hold the processes in the respective states. You may also want to have a class that represents the scheduler, which may contain a method(s) implementing your scheduling algorithm.

4. **Testing**. Test your implementation by simulating the execution of your algorithm: create a class which is capable of running your scheduling algorithm. Provide some processes to run with varying properties. For example, you may consider four processes with different execution times, and one of them having an I/O operation. Interrupts may occur during execution of processes. By executing this simulation, the output should be some text tracing the path your processes have taken through the algorithm, i.e. the different time-slices they may have, the different states they move into, etc.

Submission

Return your results to Tasks 1-4 described above, including the pseudo-code, the **Python code** and **screenshots**, in a **pdf** file on moodle, by the deadline. At the end of the first week you should be able to show and explain the analysis and the design sections tasks 1 and 2.

Add comments to your code. Place your name and student number at the top of each class file. Place comments at the method level; use one-line comments inside method bodies to describe more complex statements if you feel they are not obvious.

Tasks 1-3 have 3 marks allocated in total, while there are 2 marks allocated to task 4. Total for this lab is 5 marks.

Hints

1. You need to create the following algorithms: addReadyProcess, addBlokedProcess, addSuspendedProcess, preemptProcess, runProcess, Schedule.

2. You may use the Python *sched* module that implements a generic event scheduler for running tasks at specific times – see https://pymotw.com/2/sched/.