

COS214 Tutorial 2

Roberto Togneri, 2000

SOLUTIONS

- Give a definition of the term *process*. Explain the difference between procedure, program, process, task, and job.
 - On all current computers, at least part of the interrupt handlers are written in assembly language. Why?
 - For UNIX, give some examples of process manipulation functions.

Answer

- Definitions:
Process - instantiation of a program in the computer system; the program when it is running in the system or the activity associated with a program.
Procedure - the part of a program or region of a process which performs a well-defined function.
Program - the source or object code that resides on disk that contains the sequence of instructions to carry out a task.
Task - what you want the computer to do for you, usually when running as a process
Job - set of {programs, processes} that you want executed. Usually describes the programs not the running → submit a job... when it runs it creates one or more processes.
- Generally, high level languages do not allow one or other kind of access to the CPU hardware that is required. For instance, an interrupt handler may be required to enable and disable the interrupt servicing a particular device, or to manipulate data within a process' stack area. Also, interrupt service routines must execute as rapidly as possible.
- Process manipulation functions:
fork(): spawn a child process.
wait (): wait for the child process.
exit(): terminate a process.
setpriority(): set the scheduling priority of the process.
getpriority(): get the scheduling priority of the process.
signal(): configure the signal handler for a process
kill(): send a signal to a (another) process (e.g. SIGINTR, SIGKILL, etc.)

- The Running, Blocked and Ready states of a process imply that the process is always in main memory. However, high-level scheduling will include the ability to swap the process's main memory to disk, which is defined as the Suspend state for a process, and reload the process into main memory when it is activated.
 - Explain from which state or states a process can become suspended and why?
 - Explain the problem in deciding which state a suspended process should go to when that process is activated? What is the UNIX System V solution?

Answer

- It makes sense for the high-level scheduler to swap (suspend) processes which are wasting main memory → in the blocked state or waiting too long in the ready queue.
- Since a process was in a blocked/ready state when suspended it should go back to the ready state upon wakeup/activate. The UNIX System V solution is to create two suspend states "*Ready to Run, swapped*" and "*Sleep, swapped*"

- A blocked process still consumes system resources. Is it possible for a program to either accidentally or otherwise continually create processes that immediately block on some event that won't occur and so bring the system down?
 - If a parent process dies, what should happen to the child processes? What happens to a parent process when a child dies?

Answer

- Yes! By spawning processes and having them block you can fill up the process table making the system unusable → hit the panic button.
- If a parent process dies the child should die; however, UNIX does allow child processes to continue by giving them *init* as the parent when the real parent dies. Note: If the child process's parent loops then the child will *<defunct>* when it finishes since it is necessary for the child to return the exit status to somebody. A parent process should acknowledge that a child process has exited (via the appropriate *wait(.)* system call) otherwise the child will *<zombie>*.

- Five batch jobs *A* through *E*, arrive at a computer center at almost the same time. They have estimated running times of 10, 6, 2, 4, and 8 minutes. Their (externally determined) priorities are 3, 5, 2, 1 and 4, respectively, with 5 being the highest priority. For each of the following scheduling algorithms, determine the mean process turnaround time. Ignore process switching overhead.
 - Round Robin (RR).
 - Priority Scheduling.
 - First come, first served (FCFS).
 - Shortest job first (SJF).

For (a), assume that the system is multi-programmed, and that each job gets its fair share of the CPU. For (b) through (d) assume that only one job at a time runs, until it finishes. All jobs are completely CPU bound.

Answer

For Round Robin, during the first 10 minutes each job gets 1/5 of the CPU. At the end of ten minutes, C finishes. During the next 8 minutes, each job gets 1/4 of the CPU, after which time D finishes. Then each of the three remaining jobs gets 1/3 of the CPU for 6 minutes, until B finishes, and so on. The finishing times for the five jobs are 10, 18, 24, 28 and 30, for an average of 22 minutes. For priority scheduling, B is run first. After 6 minutes it is finished. The other jobs finish at 14, 24, 26, and 30, for an average of 20 minutes. If the jobs in the order A through E, they finish at 10, 16, 18, 22, and 30, for an average of 19.2 minutes. Finally, shortest job first yields finishing times of 2, 6, 12, 20 and 30, for an average of 14 minutes.

- Consider the implementation of all the short-term scheduling algorithms discussed in the lecture.
 - Which algorithms are effectively impossible to implement? Carefully explain why?
 - Which algorithms require a timer interrupt for the CPU?
 - Hence explain why RR is the only real scheduling algorithm that can be used in practice.
 - Can a single processor system have no processes in the ready queue? Explain what can happen in such a situation.

Answer

- (a)
- (i) RR and FCFS are easy to implement. However SJF, SRT and HRRN require knowledge of the CPU burst-time (or service time) which is theoretically impossible. Why? Even though the actual assembly instructions can be used to estimate the CPU execution cycle time, it is not possible to identify which instruction will cause a process to block since this depends on the state of the system (e.g. if data is in memory caches or buffers there is no need to block). Furthermore, the proposed aging algorithm is a major overhead and unless it is accurate (which it isn't!) will seriously undermine the effectiveness of SJF, SRT and HRRN.
- (ii) The pre-emptive algorithms: SRT and RR, require a timer interrupt facility on the CPU (early CPUs did not have this!) to allow the current process to be interrupted and the OS scheduler() to run and either select the next process to run from the ready queue or resume the currently running process (if the quantum has not expired in the case of RR, and if there are no processes in the ready queue with a shorter remaining time as in the case of SRT).
- (iii) From (i) SJF, SRT and HRRN need to be ruled out as a practical implementation is not possible. FCFS is simpler than RR but yields very poor and highly variable response and turnaround times. RR is has better response times which can be improved with priority scheduling and the response/turnaround times are not dependent on the arrival order of processes as is the case with FCFS.
- (b) Yes it can. Unless the scheduler can cope with the absence of processes the system may crash. A dummy process can be used to avoid the problem if the scheduler can't be modified. Most operating systems don't have this problem as there is always some processes running (*init*, *swapper* for UNIX).

6. (a) Explain why two-level scheduling is commonly used.
- (b) Define the difference between preemptive and nonpreemptive scheduling. State why strict nonpreemptive scheduling is unlikely to be used in a computer center.
- (c) A CPU scheduling algorithm determines an order for the execution of its scheduled processes. Given n processes to be scheduled on one processor, how many possible different schedules are there?

Answer

- (a) Two-level scheduling is needed when memory is too small to hold all the ready processes. Some set of them is put into memory, and a choice is made from that set. From time to time the set of in-core processes is adjusted. This algorithm is easy to implement and reasonably efficient, certainly a lot better than, say, round robin without regard to whether the process was in memory or not.
- (b) *Nonpreemptive*: process uses CPU until terminate or block event, next process on ready queue will then run.
Preemptive: process uses CPU and is subject to a timer interrupt to permit next process in ready queue to run.
Problems with nonpreemptive scheduling? A 1 hour CPU job will freeze the computer for other users/processes for 1 hour, not very friendly.
- (c) $n!$

7. Consider the following measured CPU service times for a process:

1 1 2 4 5 5 3 2 2 6 7 7 7

Use the aging algorithm with $a = 0.5$ to predict the next CPU service time and comment on the accuracy of this method. Assume the initial predicted service time is 1.

Answer

1:	Pred: 1	Meas: 1	Error: 0
2:	Pred: $(1+1)/2 = 1$	Meas: 1	Error: 0
3:	Pred: $(1+1)/2 = 1$	Meas: 2	Error: +1
4:	Pred: $(1+2)/2 = 1.5$	Meas: 4	Error: +2.5
5:	Pred: $(1.5+4)/2 = 2.75$	Meas: 5	Error: +2.75
6:	Pred: $(2.75+5)/2 = 3.88$	Meas: 5	Error: +1.12
7:	Pred: $(3.88+5)/2 = 4.44$	Meas: 3	Error: -1.44
8:	Pred: $(4.44+3)/2 = 3.72$	Meas: 2	Error: -1.72
9:	Pred: $(3.72+2)/2 = 2.86$	Meas: 2	Error: -0.86
10:	Pred: $(2.86+2)/2 = 2.43$	Meas: 6	Error: +3.57
11:	Pred: $(2.43+6)/2 = 4.22$	Meas: 7	Error: +2.78
12:	Pred: $(4.22+7)/2 = 5.61$	Meas: 7	Error: +1.39
13:	Pred: $(5.61+7)/2 = 6.30$	Meas: 7	Error: +0.30