



What's New:

Some answers given in tutorial mode are necessarily briefer than exam answers, because you want to check your understanding and move on.

Responses here are aimed at helping you to think through the missing details.

1. Concepts

- Why is it useful to differentiate CPU-bound and I/O-bound processes?
CPU bound processes use the CPU mostly, I/O bound processes use I/O mostly. A CPU-bound process could hog the CPU if allowed a long time slice, so it makes sense to give CPU-bound processes short time-slices. An I/O-bound process will have short bursts of CPU activity, then wait for I/O for long periods, so letting it have a longer time quantum is reasonable because it will give up the CPU soon, and do I/O again.
- Explain why starvation is a problem.
It is easy to devise a scheduling algorithm which meets general objectives, like avoiding deadlocks, and favouring small programs, but it is easy in the process to arrive at a case where some category of process will wait indefinitely, as long as other categories are available. Because it is easy to arrive at a situation where starvation occurs, yet it is difficult to define precisely, it is a difficult problem in general in OS design.
- Explain the role that context switching overheads have in designing an optimal scheduling algorithm.
Context overhead switching is effectively wasted time. Scheduling algorithms should thus try to minimise it. However, there is a tradeoff between trying to get as many jobs done and minimising context switches which should be weighed during implementation. A shorter time quantum will result in context switching being a higher fraction of total time, so scheduling methods which rely on frequent context switching (short time quanta) are likely to perform worse than expected, purely from analyzing the OS effects in the book (wait time etc.).

2. Specific Algorithms

- Which is easier to implement: shortest job first (SJF) or first come first served (FCFS)? Explain your answer.
FCFS is easier (FIFO queue). SJF also requires time estimates (as well as priority queue). Strictly speaking, true SJF can't be implemented without accurate prescience (foresight). Since in the general case, we can't know how long a process will execute, we can't implement a true SJF – hence the need to estimate the future time. But we can use SJF in simulations as a basis for evaluating less optimal algorithms.
- Why is waiting time a useful criterion to evaluate alternative scheduling algorithms?
Waiting time represents how much time a process could have been doing useful work but didn't. It is useful because small processes, with minimal waiting time, will tend to clear the system and hence free resources faster, even if total time in a short time window isn't faster. Since our back of the envelope calculations don't take into account resource conflicts, they don't capture this reason for wanting to minimize waiting time.
- Explain why preemptive SJF is better than nonpreemptive SJF. Use an example to illustrate your point.
Pre-emptive SJF is better. If a large job arrives, followed by several smaller jobs, pSJF will switch to the smaller jobs. By contrast, with nonpreemptive SJF, if a large one gets in first, the smaller ones have to wait, reducing the value of the SJF approach of finishing short ones quickly, to minimize overall waiting time.
- Explain why *exponential averaging* is a useful way to use history to estimate the length of the next CPU burst. If we used exponential averaging to calculate the overall mark for a course with several assignments and a final exam, what value of α would you prefer? Why?
The value of τ_n is the cumulative estimate. If $\alpha = 1$, this is "all exams" but $\alpha = 0$ means don't move off the original estimate. If we make the first estimate the first assignment, that will be your final result (or of course we could just give everyone the same initial mark). Try these values with different α s: first assignment used to estimate τ_0 and called t_0 result = 40, assignment 2 (t_1) = 20, assignment 3 (t_2) = 50, exam (t_3) = 50. How do results differ from doing a simple arithmetic mean?

3. Evaluation of Algorithms

- For the following scenario, draw Gantt charts for each of SJF, preemptive SJF and round robin (time quantum 10):

process	arrival time	burst time
P ₁	0	53
P ₂	10	17
P ₃	20	68
P ₄	30	24

SJF:

P ₁	P ₂	P ₄	P ₃
0	53	70	94
162			

Average = 39.25

pSJF:

P ₁	P ₂	P ₁	P ₄	P ₁	P ₃
0	10	27	30	54	94
162					

Average = 28.75

Round Robin:

Process	Execution Time	Remaining	Time (Start)
P ₁	10	43	0
P ₂	10	7	10
P ₃	10	58	20
P ₄	10	14	30
P ₁	10	33	40
P ₂	7	0 (finished)	50
P ₃	10	48	57
P ₄	10	4	67
P ₁	10	23	77
P ₃	10	38	87
P ₄	4	0 (finished)	97
P ₁	10	13	101
P ₃	10	28	111
P ₁	10	3	121
P ₃	10	18	131
P ₁	3	0 (finished)	141
P ₃	10	8	144
P ₃	8	0 (finished)	154
			162

Average = 60.5

Worth showing the detailed working in case you have a number wrong (e.g., in SJF, average waiting time = $(W(P_1) + W(P_2) + W(P_3) + W(P_4))/4 = (0 + 43 + 74 + 40) / 4 = 39.25$, where the waiting time in each case = start time - arrival time. To avoid making a mistake, it is worth marking the time when each process arrives (especially for round robin, where you have a lot of numbers to calculate).

- b. Based on answers of (a), which algorithm gives the shortest average wait time? Given the complexity of estimating how long the next CPU burst is, would you choose SJF over round robin, given the data here? What might we be missing which could change the answer?

Preemptive SJF is the obvious answer. The very big difference in both SJF variations and round robin appears to imply that round robin is a poor strategy. Clearly it can't be that bad since it's actually used. The missing issue is responsiveness. By setting the time quantum to a good compromise value, approximately the maximum needed by the majority of processes, but not so big as to make interactive processes unresponsive, you can get a better compromise between responsiveness and minimum wait time than in this example.

- c. In a proposed scheduling scheme, all processes start with priority 0, which means they can be processed immediately if ready and at the head of queue 0. If they fail to complete a time quantum (10 time units), their priority is reduced to 1 (in this case, bigger numbers are worse) and they drop to queue 1. Queue 1 processes are never scheduled until there are no ready queue 0 processes. A queue 1 process has a time quantum of 20, and is promoted to queue 0 if it does not complete its time quantum (e.g., because it has to request I/O).

- Which of the previously considered schemes in this question does this approach *most closely* resemble?
pSJF to a rough approximation, if the time quantum is set to that needed by the least CPU-bound processes.
- Considering all schemes covered in lectures, what would you call this proposed new scheme?
Multi-level feedback is pretty much defined the same way.
- What major flaw does it have, and how could you fix the flaw? Describe your improved version.
Long CPU bound processes can suffer starvation, but this can be fixed with aging (processes which are not scheduled have their priority increased)
- Redo part (a) with the variations in this question (original and your improved version).
To answer this would take a fair amount of work: worth saving up as a revision pre-exam question.

- d. Consider values of $\alpha = 0, 1, 0.25$ and 0.5 . How does the value of τ vary with each value of α , with $\tau_0 = 32$, and successive values of $t = 16, 12, 8, 4, 100, 100$? (If you have time, draw a graph in the style of Figure 6.3, p 160 for each value of τ . Note that the horizontal axis of the graph represents time quanta.)

In general,

$$T_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$$

$$\alpha = 0, \tau_n = \tau_0 = 32$$

$$\alpha = 1, \tau_{n+1} = t_n$$

$$\alpha = 0.25, \tau_1 = 0.25 \times 16 + 0.75 \times 32 = 28; \text{ continue in the same way to get successive values: } 24, 20, 16, 37, 52.75$$

$$\alpha = 0.5, \tau_1 = 0.5 \times 16 + 0.5 \times 32 = 24; \text{ continue in the same way to get successive values: } 18, 13, 8.5, 54.25, 77.125$$

For $\alpha = 1$, history is ignored, i.e., the latest estimate is always the last observed time. For $\alpha = 0$, the guess is not varied from the initial guess. The simplest solution in the other cases would be to draw a table (or cheat: use a spreadsheet) with one entry for each value of τ_i . Some explanation is called for to make it clear what you are doing here. It would be a useful exercise to draw the graph, but not too hard given the numbers.

4. Discussion Questions

- a. Can a real-time scheduler can be implemented in conjunction with a general-purpose operating system? Discuss.
A kernel needs to support real-time specifically if hard real-time requirements are to be met. It has to be possible to specify a maximum permissible turnaround time for a realtime process, including scheduling overhead, and the realtime process has to

complete in that time, otherwise an error has occurred. In exchange for guaranteeing to complete in that time, the OS will not interrupt the realtime process until the end of the time it has been guaranteed. Without this kind of support in a kernel, hard-realtime can't be implemented, because interrupts in general can't be turned off in a user-level process without some safeguard to ensure that the OS can get control back. In a dedicated real-time OS, where no other processes may be running (or of significance), the issues may be different.

- b. Multilevel feedback queues are a mechanism to balance throughput with responsiveness. Discuss how they can achieve both goals, looking at any real operating system of your choice.

Worth answering as a revision question later. Look up the discussion of Windows NT and successors in the text book.

- 5. Do the questions in the week 4 classroom exercise (available in [PowerPoint](#) and [PDF](#)) if you didn't do them already.

Worth doing especially if you weren't at the lecture. If you have an answer on which you would like comments, let me know.