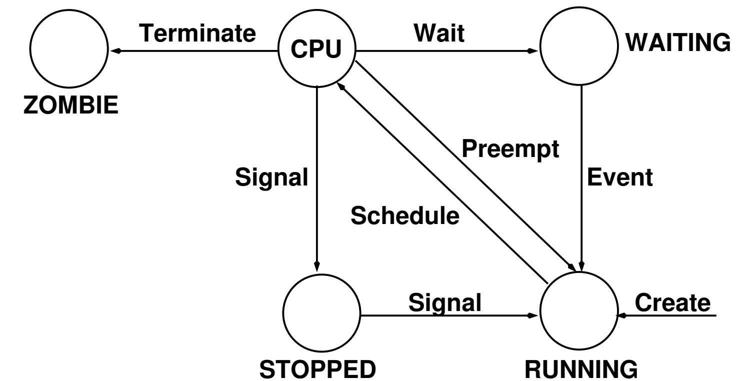
Overview

- We looked earlier at the process life-cycle and saw that over its lifetime a process moves between various states
- At any instant a number of processes may be in a runnable state, in the run queue ready to do work if given the CPU
- It is the scheduler's job to choose which process to move next from the run queue to the CPU
- We look at some common scheduling algorithms before concluding with a brief description of scheduling in Linux
- Depending on the context some algorithms are more suitable than others e.g. Linux is an interactive operating system and so attempts to minimise response times

The Process Lifecycle

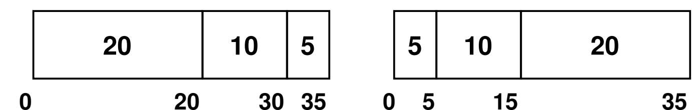


First-Come, First-Served

- Under this approach processes are simply scheduled in the same order in which they arrive at the run queue
- Once a process is assigned the CPU it retains it until it terminates or blocks (e.g. on an I/O operation)
- There is no preemption i.e. a process must voluntarily relinquish the CPU
- FCFS is the simplest of scheduling algorithms to implement but performs poorly if short processes get stuck behind a long one: a convoy effect arises

First-Come, First-Served

- Under an FCFS scheme the average response time (ART) and average completion time (ACT) can vary considerably depending on the arrival order of processes at the run queue
- ART: $(0 + 20 + 30)/3 = 16.7$ vs. $(0 + 5 + 15)/3 = 6.7$
- ACT: $(20 + 30 + 35)/3 = 28.3$ vs. $(5 + 15 + 35)/3 = 18.3$



Round Robin

- To get around the problem of being stuck behind a slow coach and to make sure every process makes some progress we can assign to each a quantum/timeslice
- After its quantum expires a process is removed from the CPU and moved to the tail of the run queue and we thus ensure CPU time is fairly distributed across processes
- Too short a quantum and the percentage of time spent context switching (this is pure overhead where no progress is being made) becomes exorbitant
- Too long and response times suffer and we revert to FCFS
- We still have no notion of process priority however. . .

Shortest Job First

- Looking back at the example met earlier it seems that if a scheduler could tell for how long a process would require the CPU it could do a good job at minimising response times if it ran processes in shortest-first order
- It turns out that ordering jobs shortest-first is provably optimal in terms of minimising response times
- Is minimising response times something an operating system like Linux would like to do? Should Linux employ an SJF algorithm?
- Well, every process will run for some period (a CPU burst) before issuing an I/O request that causes it to block

Shortest Job First

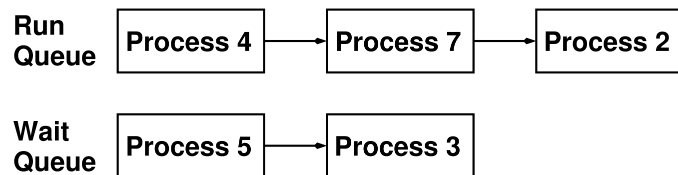
- If we employed SJF scheduling on Linux then processes with short CPU bursts will run ahead of others
- These (interactive) processes are the ones that are regularly carrying out I/O operations e.g. your word processor, your MP3 player, the shell where you type commands etc.
- It turns out that these are precisely the processes to which we would like to give priority e.g. when you hit a key you want a rapid response from your word processor
- Compare with a (non-interactive) process calculating π, you are not too bothered if it takes an extra couple of seconds to complete

Shortest Job First

- So applying SJF seems a good option since it prioritises interactive processes but implementing it is not straightforward since we cannot tell in advance how long a process's next CPU burst will last
- We can however achieve a similar effect if we monitor a process and based on its interactivity assign it a priority
- The more interactive a process, the more I/O it carries out and the more it sleeps waiting on I/O results and we can measure for how long a process sleeps on average
- By prioritising processes based on their interactivity we are approximating shortest job first scheduling

Priority Scheduling

- We keep processes ordered by priority on a queue and, once priorities are assigned, preemption becomes possible
- We run a process until its quantum expires, it blocks or the new head of the runnable queue is of a higher priority than the currently executing process
- Keeping the queue in order introduces overhead and is somewhat inflexible so we have one final step to take. . .



Multilevel Priority Queues

- Linux uses multiple queues to implement priority scheduling with one queue per priority level
- Processes are scheduled from the highest priority non-empty queue and moved to the tail of the corresponding queue when rejoining
- New processes are put on a medium priority queue until their interactivity can be established
- Processes move between queues as their priority varies (sometimes a process starts off interactive but changes)
- Different scheduling algorithms can even be employed across the various priority queues

Multilevel Priority Queues