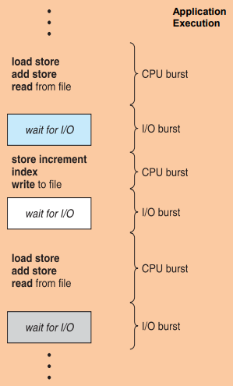
Operating Systems

Scheduling

Scheduling

Scheduling is the process of deciding what will run next, often picking among the processes or threads in the ready queue(s). These decisions are potentially made whenever we switch to the OS due to an interrupt, syscall or exception.

The decisions are made by a scheduler according to a policy (implemented by an algorithm) and the tasks are then switched by the dispatcher, either a kernel or user-code mechanism for doing so.

Process or Thread Behaviour

Process or thread execution consists of cycles of CPU execution (CPU burst) and I/O waiting (I/O burst). The distribution of these bursts is application dependent.

With multiprogramming we want to achieve maximum CPU utilisation so we don’t leave the CPU idle and waste time. To do this, while a process or thread is waiting for I/O another can run on the CPU (focusing on a single CPU/core).

Processes and threads can be CPU bound or I/O bound, meaning that they favour CPU or I/O bursts. I/O bound processes and threads are good for multiprogramming as while they wait for I/O another can run on the CPU.

Scheduling Goals: Performance

There are many performance goals (which may conflict) for scheduling to achieve:

* Maximise CPU utilisation
* Maximise throughput (processes completed per time unit)
* Minimise turnaround time (time from submission of tas to completion)
* Minimise waiting time (all periods spent waiting in the ready queue from submission)
* Minimise response time (time from submission of request to response being produced)
* Minimise energy (joules per insturction) subject to some constraint (e.g. frames/second)

In most cases we otpimse the average metric (but sometimes we want to minise the worst case).

Scheduling Goals: Fairness

There is no single compelling definition of fairness, how should be measure it? Equal CPU consumption? (over what time scale?) fair per-user? Fair per-process? Per-thread? What if one process is CPU bound and one is I/O bound?

Sometimes the goal is to be unfair to explicitly favour some particular class of requests (a priority system) to avoid starvation making sure everything gets at least some CPU time.

Classes of Schedulers

Batch

* Throughput / utilisation oriented
* Example: Pixar’s rendering

Interactive

* Response time orientated
* Example: window-based operating system

Real-time

* Deadline driven
* Example: embedded systems (cars, airplanes etc…)

General Scheduling Problem

Who to we assign each resources to and when should we re-evaluation our decisions?

Scheduling is not just about assigning processes and threads to the CPU, but to any other HS/SW resources of the computer too.

When to Re-Evaluate the Decision

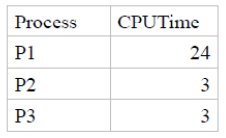
There are two general approaches, non-pre-emptive and pre-emptive scheduling.

Non-pre-emptive scheduling allows processes/threads to execute until completion or until they want using voluntary process switches and switches on blocking calls. The scheduler gets involved only at exit or on request.

Pre-emptive scheduling may pause a process/thread while it’s executing then resuming a different process/thread. These are known as involuntary process switches, for every clock interrupt, running processes may be suspended and switched with another process (if there are any).

Scheduling Algorithms

First-Come First-Served (FCFS)

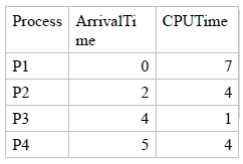
Process are assigned to the CPU in the order they were requested (or the order the requests arrived in). This is non-pre-emptive so each process must wait for all of the process before it to finish.

The order in which processes arrive in FCFS has a large effect on the average turnaround time for example if the order is P1, P2, P3 the average turnaround is 27, however if it is P2, P3, P1 the average turnaround is 13.

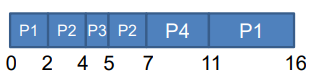
The average response time for FCFS can be very poor having short tasks wait behind large ones (convoy effect). This may lead to poor utilisation of other resources too (poor overlap of CPU and I/O activity).

Shortest Job First (SJF)

We associate each process with the length of its CPU time, we then choose the process with the shortest CPU time to run first.

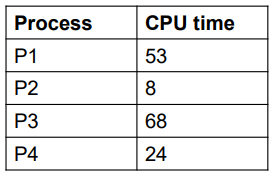
There are two variations of this algorithm, non-pre-emptive where once the CPU is given to the process it cannot be taken away until completion (or blocking) and pre-emptive where if a new process arrives with CPU time less than the remaining time of the currently executing process we pre-empt it.

For non-pre-emptive SJF, the example at the right would have an average turnaround time of 8 with 3 context switches.

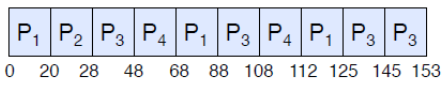
For pre-emptive SJF, the example would have an average turnaround time of 7 with 5 context switches.

It can be shown that pre-emptive SJF is optimal but it can only ever be approximated as it’s too complex to be implemented in practice and it’s not always possible to determine the CPU/I/O burst lengths. SJF can also result in starving processes as new processes coming in could always be shorter than it.

Round-Robin (RR)

Each process is allowed to run for a specified time interval (called a quantum), after this time has elapsed the process is pre-empted, added to the end of the ready queue and the next process is scheduled. If the process terminates or blocks for I/O before this time it is added to a wait queue and the next process is scheduled.

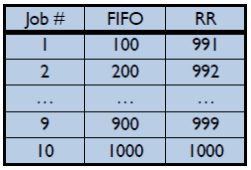
In the example to the right the average waiting time will be 66.25 and the average turnaround time will be 104.5.

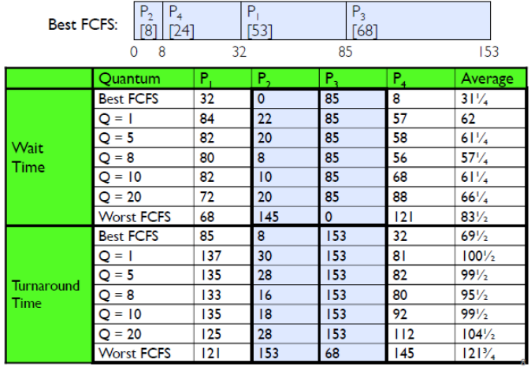


Context switching may impact the choice of the time quantum. For example if a context switch takes 1ms and the time quantum is 4ms then 20% of our time is thrown away context switching.

Context switching is typically in the order of tens of µs and the timeslice/quantum is 1KHz (every 1ms). But when there are lots of processes, a long time quantum can causes a poor response time.

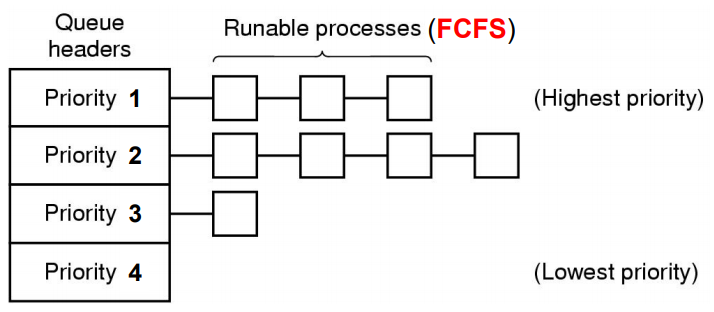
RR is a solution to fairness and starvation as there’s a fair allocation of the CPU across all the jobs, it also has a low average wating time when job lengths vary and it’s good for responsiveness (interactivity) if there are a small number of jobs. However the context switching time may add up for long jobs.

FCFS Vs RR

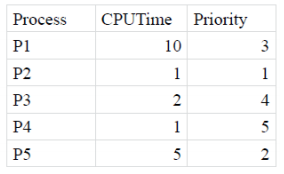
Assuming zero-cost context-switching time, is RR better than FCFS?

Here we have 10 jobs, each taking 100s of CPU time, assume RR has a quantum of 1s and all jobs start at the same time.

Both RR and FCFS finish at the same time but the average turnaround time is much worse under RR (bad when all jobs have the same length). Additionally cache state must be shared between all jobs, this may slow down RR making the total time for RR longer even for zero-cost switch.

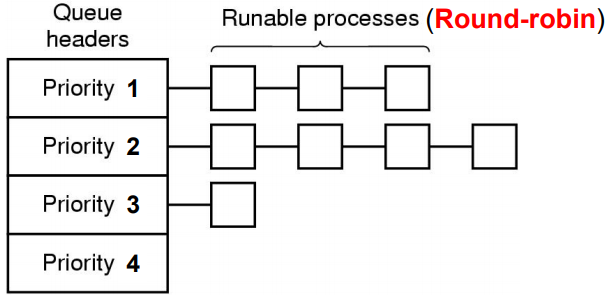
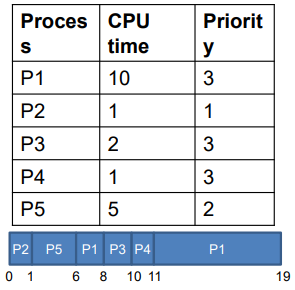
Priority (PRIO)

Priority scheduling uses different classes of priority to determine the order in which process should be executed with higher priority processes being run before lower priority ones (often in FCFS).

Considering the example to the right, the average turnaround time would be 12.

This system often suffers from starvation issues where lower priority jobs don’t get to run because higher priority tasks are always running and being added. There is also the issue of deadlock or priority inversion, this happens when a low priority task has a lock needed by a high priority task, as such the high priority task can’t complete until the low priority task has released the lock, but the low priority task can’t run and release the lock until the high priority task has completed. This isn’t strictly a problem with priority scheduling though.

How to Assign Priorities

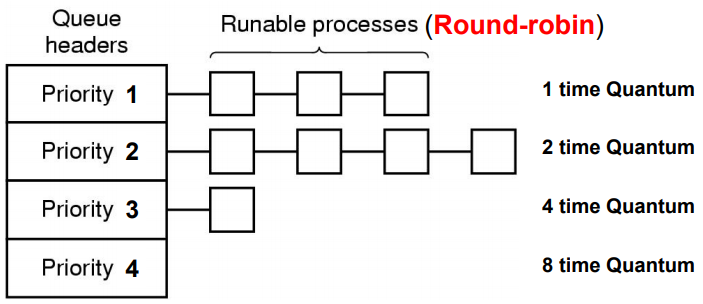
We can assign priorities statistically based on process type, the user and how much the user pays to use the system or dynamically based on how much the process runs vs how much it uses I/O with priority being 1/f where f is the size of the quantum used last (the longer a process ran, the lower its priority).

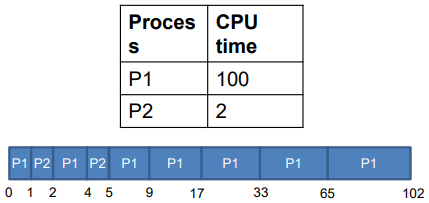
Multiple Queues

Multiple queues is almost the same as the priority method except each level uses round-robin instead of FCFS.

Considering the same example as the priority with a time qantum of 2, the average turnaround time for this case would be 9.4.

Multilevel Feedback Queue (MLFQ)

This is the same as MQ scheduling but each queue has a different time quanta with shorter quanta for higher priority. Every process starts at the highest priority, when a process exceeds its quanta it’s moved to a lower priority, when a process becomes interactive it’s moved to higher priority.

An issue with this is that if the user discovers how to make their tasks interactive, they can play the system.

From the example to the right, the average turnaround time will be 52.5 with 8 context switches. This contrasts the 101 switches which would have been required with fixed quantum.

Algorithm Evaulation

How do we select a CPU-scheduling algorithm for a system? We do so based on criteria/a goal which could be any of the goals mentioned earlier (CPU utilisation, response time, etc…).

Methods for evaluating each algorithm include:

* Deterministic modelling (analytical evaluation), given an algorithm and the (known) system workload, evaluate the performance
* Queueing models (mathematical model)
* Simulation (programming a model)
* Actual-implementation (real-world testing)

Multicore/Multiprocessor Scheduling

Multicore/multiprocessor scheduling is more complex as they have to take into account:

* Multicore CPUs
* Multithreaded cores
* NUMA systems
* Heterogeneous multiprocessing
* Load-sharing (processes/threads can run in parallel)
* Asymmetric multiprocessing (AMP) which is common in embedded systems
* Symmetric multiprocessing (SMP) which is widely adpted (Linux, Windows, etc…) all processes and threads are in a common ready queue with each processor having its own ready queue and using work sharing/stealing.