Lecture 6 - Scheduling part 1

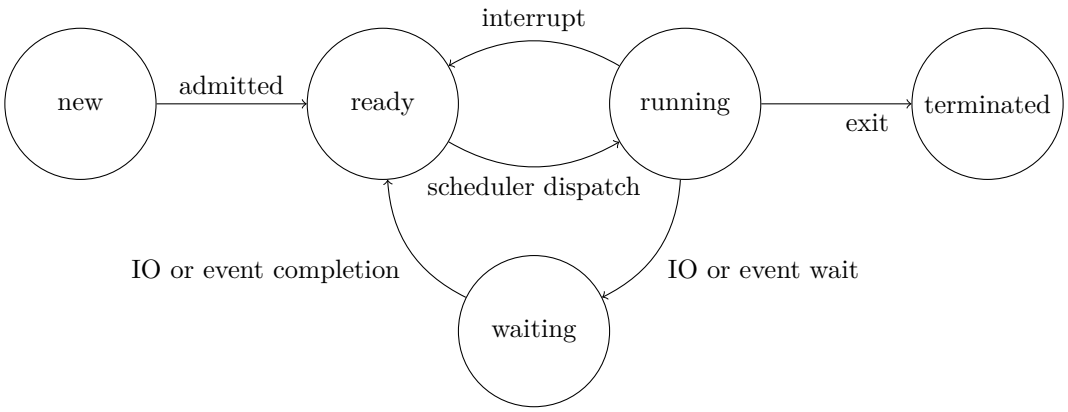
Gidon Rosalki

2025-05-04

Notice: If you find any mistakes, please open an issue at `https://github.com/robomarvin1501/notes_operating_systems`
Schedulers are everywhere, any resource with multiple users requires a scheduler, and accordingly we have many different types of scheduler:

- CPU scheduler (short term scheduler)
- Mid/long-term scheduler (loading into memory)
- Scheduling jobs in the print queue
- Scheduling I/O device requests (e.g. disk scheduler)
- Packet schedulers when networking, deciding which packet is sent in what order
- Scheduling requests in web servers
- Even scheduling in a supermarket / post office, deciding which customer is served first

Recall the process state diagram:



This diagram makes use of blocking processes, and the scheduler brings them back to life, but until now we have not so much discussed the why or how either of these work. Which process should run, and for how long on the CPU?

To decide this question, we have the scheduler, and the dispatcher. The CPU **scheduler** decides which process should run on the CPU (and sometimes for how long). This is a short term scheduler, because it runs tens/hundreds of times a second. The **dispatcher** is the module responsible for executing the CPU scheduler decisions. It handles the context switch, and returning to user mode.

We will assume that different placements of processes by the scheduler are always different jobs, which terminate when the process is blocked, for simplicities sake. Interactive (reactive) programs have multiple CPU bursts, one after each event, or perform I/O operations separated by CPU bursts.

A scheduler is an algorithm:

- **Input:** The jobs to schedule
- **Output:** A decision which job to run now
- **Objective:** "Good performance" (a beautifully nebulous term, that enables even the worst schedulers to be considered good)
- **Available actions:** Can we pre-empt?

We actually have many different objectives for a scheduler. We want a low response time, ie a low total time to complete a job (wait time + run time), since this is what users care about, though not everything is in the scheduler's control. The wait time **is** what the scheduler controls, and the slowdown is how much slower the system appears to be, and is measured by:

$$\frac{\text{response time}}{\text{run time}}$$

We also want there to be a high throughput, which is measured as

$$\frac{\#\text{completed jobs}}{\text{time unit}}$$

If this system is stable, then this is equal to

$$\frac{\#\text{started jobs}}{\text{time unit}}$$

For today, we will assume that our systems are stable.
We also want high CPU utilisation, i.e. a high fraction of time that the CPU is busy, not counting overhead. This is highly affected by the availability of jobs, and both limited when the system is overloaded.
We also want to ensure fairness, and give each user their fair share, avoid starvation (less of a problem on a stable system), and support user/job priorities (ah, but who sets these priorities?). We will typically assume equal shares and priorities.
The objectives might contradict each other, in order to optimise throughput, we run jobs to completion, which reduces overhead, since we waste less time on context switches, but to optimise response time, we must schedule each new job *as soon as possible*, even if this leads to overhead due to context switching. Thus in each case we need to ask what are the right objectives for me at this time?

1 Offline algorithms

An offline algorithm gets all the input at the outset. This is the "easy" setting. Online gets the input piecemeal, meaning that at every instant you only receive part of the input. You thus need to produce output based off your current knowledge, and you may come to regret those decisions later.

We shall use the system model where all jobs are given in advance, with no additional arrivals, and the job runtimes are known in advance.

1.1 Baseline scheduler: FCFS

Here we will use a first come first serve algorithm, consider

Job	runtime
P1	17
P2	2
P3	3

Table 1: Jobs to schedule

So this will run P1, P2, then P3, taking $17 + 2 + 3 = 22$. From this we can create the wait time-response time table:

Wait time	Response time
0	17
17	19
19	22

Table 2: Wait time-response time

So our average waiting time is $\frac{0+17+19}{3} = 12$, average response time is $\frac{17+19+22}{3} = 19.33$, and our throughput is $\frac{3}{22} = 0.136$. This is overall not great, all our tasks wound up waiting a long time to execute. We call this the convoy effect: short jobs get stuck behind long jobs.

1.2 Shortest job first: SJF

We can improve this by putting the short jobs before the long jobs. Should we have 2 jobs, p1 that takes 4, and p2 that takes 7, if we run p2 first, then our average wait time is 3.5, and our average response time was 9, but if we swap them them then our average wait time is now 2, and our average response time is 7.5. However, the throughput remains unchanged at 0.182. Note, that this applies **every** time a short job comes after a long one.

Let us once again consider

Job	runtime
P2	2
P3	3
P1	17

Table 3: Jobs to schedule

then so this time:

Wait time	Response time
0	2
2	5
5	22

Table 4: Wait time-response time

and our new average wait time is 2.33, new average response time is 9.67, though the throughput remains constant at 0.136.

1.3 Optimality

Can we improve this? No. SJF is optimal in the offline setting, where it aims to optimise the **average wait time**. Pre-emtpion does not help and the average response time is also optimal. The core of the proof is as follows:

1. Assume a schedule S has the optimal average wait time
2. If S is not SJF, then there are a pair of jobs such that p_i.runtime $> p_{i+1}$.runtime
3. Switching these jobs reduces their contribution to the average, without changing anything else
4. This is a contradiction to the assumption that S is optimal

Hey look! This is just like proving greedy algorithms from algo. It is left as an exercise to the reader to turn this into a real proof.

2 Online algorithms

Here jobs will arrive at unknown times (this is called an open system model), but job runtimes are known in advance. We will also say that there is no pre-emption.

2.1 FCFS

Consider the following input:

Job	Arrival	Runtime
p1	0	7
p2	2	4
p3	3	2
p4	6	3

Table 5: Job arrivals

Wait	Start	End	Response
0	0	7	7
5	7	11	9
8	11	13	10
7	13	16	10

Table 6: Wait time-response time

This is still a FCFS method for scheduling, just in an open system. The average wait here is 5, with an average response of 9, and a throughput of 0.25. The question is if we could do better, consider for how long every subsequent job had to wait before it could begin.

2.2 SJF

Consider the following input:

Job	Arrival	Runtime
p1	0	7
p2	2	4
p3	3	2
p4	6	3

Table 7: Job arrivals

Wait	Start	End	Response
0	0	7	7
10	12	16	14
8	7	9	6
3	9	12	6

Table 8: Wait time-response time

Here we have slightly improved things, our average wait time has dropped to 4.25, the average response to 8.25 (and naturally the throughput remains unchanged at 0.25)

2.3 Priority scheduling

Many scheduling algorithms can be interpreted as priority scheduling algorithms, we assign each job a priority, and then schedule the job with the highest priority. In FCFS the priority is the time since arrival, and in SJF the priority is the inverse of the runtime.

SJF had a problem, like every online algorithm, it does not know the future. When p1 arrived it was scheduled, but then the scheduler was stuck until p1 terminated, even when shorter jobs became available. The solution here is to use preëmption, we revert earlier decisions that turn out to be suboptimal. This compensates for not knowing the future, *however* comes at the cost of the overhead associated with context switching.

Let us alter our model, such that we may also use preëmption:

Job	Arrival	Runtime
p1	0	7
p2	2	4
p3	3	2
p4	6	3

Table 9: Job arrivals

Wait	Start	End	Response
(9)	0	16	16
(2)	2	8	6
0	3	5	2
2	8	11	5

Table 10: Wait time-response time

Where the wait in brackets indicates the time spent suspended by a preëmpt. So the average wait is down to 3.25, with an average response down to 7.25, and the throughput once again remains the same at 0.25.

3 Not knowing runtimes in advance

Now for the most realistic model: Jobs arrive at unknown times (open system model), job runtimes are **not** known in advance, and we can use preëmption. So the problem here is that we still don't know the future (shocking, I know), and when a new job arrives we do not know whether it will be short or long. We thus do not know whether or not to preëmpt, and thus short jobs can get stuck behind long jobs. To solve this we estimate. We assume that a job has multiple CPU bursts, and use the exponentially weighted average of previous bursts to estimate the length of the next burst:

1. t_n is the actual length of the nth CPU burst
2. τ_{n+1} is the predicted value for the next burst
3. $0 \leq \alpha \leq 1$ is the weight factor
4. We define $\tau_{n+1} = \alpha t_n + (1-\alpha)\,\tau_n$ (where $\tau_1 = 0$)
5. Result: $\tau_{n+1} = \sum_{i=1}^{n} (1-\alpha)^{n-i}\,\alpha t_i$

The motivation here is that if a process starts taking too long to run, we assume that it will continue so to do, and so context switch. This might not give good results, and has the additional problem of lots of overhead.

3.1 Example for estimate

Let us define $\alpha = \frac{1}{4}$, and $\tau_{n+1} = \sum_{i=1}^{n} \left(\frac{3}{4}\right)^{n-i} \cdot \frac{1}{4} t_i$:

Time step	Actual	Next step (prediction)
1	$t_1 = 10$	$\frac{1}{4} \cdot 10 = 2.5$
2	$t_2 = 1$	$\frac{3}{4} \cdot \frac{1}{4} \cdot 10 + \frac{1}{4} \cdot 1 = 2.125$
3	$t_3 = 1$	$\left(\frac{3}{4}\right)^2 \cdot \frac{1}{4} \cdot 10 + \frac{3}{4} \cdot \frac{1}{4} \cdot 1 + \frac{1}{4} \cdot 1 = 1.843$
4	$t_4 = 1$	$\left(\frac{3}{4}\right)^3 \cdot \frac{1}{4} \cdot 10 + \left(\frac{3}{4}\right)^2 \cdot \frac{1}{4} \cdot 1 + \frac{3}{4} \cdot \frac{1}{4} \cdot 1 + \frac{1}{4} \cdot 1 = 1.632$

Table 11: Predicted time steps

3.2 Alternative partial solution

Let us assume that jobs can run together (by splitting apart the CPU into many parts, each able to run a job). This is called processor sharing, and when there are k jobs, they all advance at a rate of $\frac{1}{k}$.

Job	Arrival	Runtime
p1	0	7
p2	2	4
p3	3	2
p4	6	3

Table 12: Job arrivals

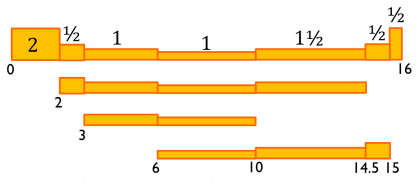


Figure 1: Processor sharing visuali-

Wait	Start	End	Response
0	0	16	16
0	2	14.5	12.5
0	3	10	7
0	6	15.5	9.5

Table 13: Wait time-response time

The average wait is undefined, with an average response of 11.25, and the throughput is still 0.25. This is good since short jobs do not get stuck, however it has the drawbacks of every process running at a slower rate. So is it really beneficial? Well, this depends on workload statistics, specifically on the distribution of job lengths. Should we have 3 processes, 1 very long, and two very short, then this is the good case, and it even approximates SRPT. However, should all the tasks require the same amount of time, then all the jobs will take that amount of time longer (4 jobs of equal length all take 4x longer), and is even worse than FCFS. Remember, this was currently a theoretical discussion, since we cannot actually share our processor. So in theory, processor sharing is good for **skewed** distributions, where there are many small values, and few large values. Specifically, this is good when $CV > 1$, where CV us the Coefficient of Variation

$$CV = \frac{\sigma}{\mu}$$

(ie, standard deviation over the mean).

So, some people (citation needed) measured lots of things, and found most processes are indeed very short, though some are very long, and the distribution has a pareto tail:

$$Pr\left(r > t\right) = \frac{1}{t}$$

Though with the caveat that there is little data, and different systems are probably very different. Overall however, it does appear that processor sharing should be really rather good, such a shame that it cannot be done.

3.3 How to do processor sharing

OK, this title is misleading, but we can approximate it. Consider the round robin scheduler (implemented in exercise 2). We run each process for a certain time quantum (predetermined short amount of time), and then preëmpt, and move the current process to the back of the queue. Let us return to the above examples:

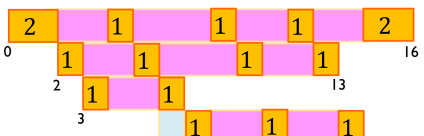
Job	Arrival	Runtime
p1	0	7
p2	2	4
p3	3	2
p4	6	3

Table 14: Job arrivals

Wait	Start	End	Response
(9)	0	16	16
(7)	2	13	11
(2)	6	7	4
(5)	7	14	7

Table 15: Wait time-response time

Average wait: 5.5, average response: 9.5 (up from 7.25 in SRPT), throughput: 0.25 So as we see, it is an improvement, if not for the massively increased context switching.



However, we still need to choose our quantum. With n jobs, and quantum q, no job waits more than $(n-1)\,q$, and thus a shorter q, results in a shorter wait.

However, this has the cost of context switches. If a context switch takes time c, then the overhead will be $\frac{c}{q+c}$, so a shorter q results in more overhead, the smaller q results in an overhead that tends to 1. If q is too long, then bursts will finish before the end of the quantum, and this will lead to behaviour like FCFS. We have empirical data on effective quanta. The attached table has data on the distribution of actual running time for different apps (till finished burst, external interrupt, or end of quantum).

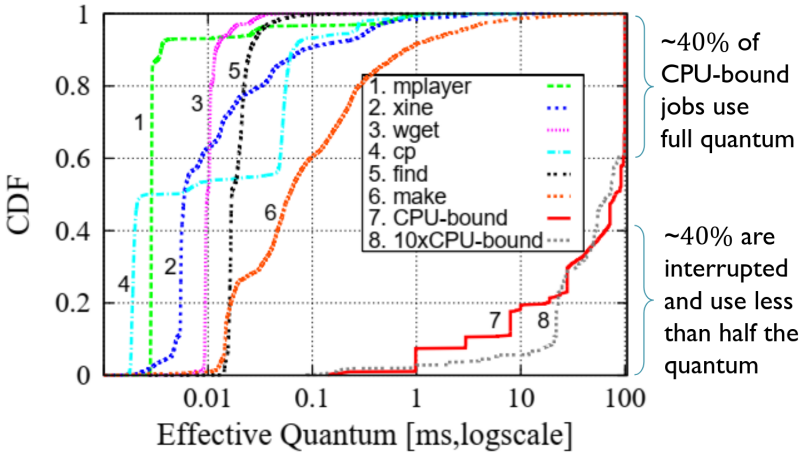


Figure 3: Effective quanta

It should also be noted from this table that multimedia apps use less that $\frac{1}{1000}$ of the effective quantum.

To implement RR, the scheduler selects a process to run, and then gives it to the CPU. The OS then loses control of the system, and possibly will not regain control until a system call. So how can we enforce a limited time quantum? Through hardware support of periodic clock interrupts. This the only way for the OS to regain control over the CPU, and once the OS is in control, it can perform a context switch.

So RR works in an online setting, and uses preëmption to cope with a lack of knowledge of when additional jobs will arrive, and for how long jobs will run. RR gives uniform treatment to all jobs, and we can theoretically do better, and this is left as an exercise to the reader :)