Tutorial 6 - Scheduling

Gidon Rosalki

2025-05-07

Notice: If you find any mistakes, please open an issue at `https://github.com/robomarvin1501/notes_operating_systems`

As opposed to lots of mathematical courses, this tutorial will mainly be backed by heuristics, as opposed to actual proofs that certain things are optimal. We will prove that some things are more optimal than others, but not that one heuristic is more optimal than another.

1 Ex3

In Ex3 we will implement a MapReduce library. The map reduce pattern is very common, and is made of a combination of the `map` and `reduce` concepts. `map` applies an independent function over an input, consider mapping a function that squares the input to an array of numbers, each element can be applied independently. `reduce` is a function that gathers the result by an accumulative function, for example `sum`. The important concept here is that we can use *parallelism* to accelerate this process.

Consider the example of counting how many times each letter appears in a file. We can do this in the following stages:

- Preprocessing: Get all the words of the file
- Map: Count how many times each character of each string appears in it
- Sort: For each possible character, get all the counts from all strings
- Reduce: For each character, sum the number of appearances

It is important to note that between Map and Sort there is a barrier, the sort needs to wait for all the chatacters to have been counted, and before we can reduce, we need to wait for the sort to finish.
To achieve this the file is split into substrings, each substring has its letter frequencies counted, then the frequencies are sorted (so all the 'a's are together, and all the 'b's, and so on), and then in the reduce phase, we can sum together all the relevant frequency counts.

This year, we will be using C++ threads, rather than C posix threads, so the solution will be rather different in comparison to previous years.

2 CPU scheduling

2.1 Introduction

Scheduling is to determine which process runs in each one of the current CPUs (consider the library in exercise 2, it used a round robin scheduler). The scheduling is the job of the operating system. The OS scheduler decides what job to run, and the dispatcher is responsible for making context switches, i.e. switching between the processes, which is carrying out the commands of the scheduler.

We want to maximise **utilisation** - i.e. the CPU time of "relevant" instructions (not context switching), and keep the CPU as busy as possible. We also want to maximise **throughput**, i.e. maximise the number of completed processes in a time unit. We want to minimise **waiting time**, the amount of time a process has been waiting in the ready queue. Finally, we also want to minimise **turnaround time**, the amount of time needed to execute a particular process.

Scheduling is difficult! To describe things formally, each process has a requested running time. We do not know the running time, and sometimes cannot even estimate it, and we can not assume that the processes arrive at the same time. Even if these two assumptions are true, scheduling is NP-hard, given two CPUs or more. We will discuss one CPU later. Since it is NP-hard, we will need heuristics (i.e. we have no formal proof, but by running lots of experiments, we can say that one thing is better than the other).

The scheduling algorithms have some properties. All the processes should have **fair** use of the CPU. We need to avoid **starvation**, which is caused when a process can never get to use the CPU, for some reason (consider one thread always block another). **Preëmption** is whether or not a process can be stopped in the middle of execution, non voluntarily.

Offline/online - An offline algorithm gets all the input at the outset. 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.

There are some different types of scheduling.

- Batch: The jobs need not the intervention of the use, for example training a neural network. The OS is the thing that schedules the process, and the only thing that matters is the completion of the jobs.
- Interactive: Users are supplying input, and output, so the scheduling needs to focus on responding quickly to the user input (consider using a web browser / IDE)
- Real Time: The execution should be completed before a given time (a deadline). RTOSs are generally used in industrial contexts, or cars, where it needs to be certain that tasks will finish in set amounts of time.

2.2 Algorithms

2.2.1 Measurement

We measure the performance as follows:

- CPU utilisation is $\frac{\text{Actual running time}}{\text{Total running time}}$. Note, this includes the time for context switches
- Average turn around time: $\frac{\text{Sum of time spent running jobs and context switches}}{\text{Number of jobs}}$
- Average waiting time: $\frac{\text{Time spent waiting for each job to start running}}{\text{Number of jobs}}$
- Throughput: $\frac{\text{Finished jobs}}{\text{Total running time}}$

2.2.2 First-Come First-Serve (FCFS)

The process that asks for the CPU first, gets the CPU first. This is the simplest scheduling algorithm, and can be implemented using a queue. New tasks are pushed to the tail, and popped from the head. This algorithm is starvation free, everything will eventually complete, and thus everything will eventually be reached, however, the wait times can be very long.

2.2.3 Shortest Job First/Next (SJF/SJN)

When the CPU is available, it is assigned to the process that will take the shortest amount of time. This can be proved to be optimal for waiting time, in the offline context, which is when the jobs all arrive at time 0, and when we know how long each job will take. This has the problems that we need to know in advance the job execution time (we often do not), and that this has starvation in the online settings (consider we have a long job in the queue, but just before the first short job stops, a new short job arrives, then the long job will never run).

2.2.4 Shortest Remaining Time First (SRTF)

Often called Shortest Remaining Time. This is a preëmptive variant of SJF. If a new process arrives with a required CPU time which is smaller than the remaining time of the current process, current executing process, then preëmpt it. This way, short processes are handled very quickly. However, this has the same problem as SJF, in that job execution times are not necessarily known, and it can cause starvation.

2.2.5 Priority scheduling

The CPU is allocated the process with the highest priority. A priority is associated with each process, each priority is a value in a fixed range of numbers, where we use **low** numbers to represent **higher** priorities. Processes with the same priority are ordered in FCFS. Note that SJF is a special case of this, where the priority is the time required on the CPU. This has the problems of not being starvation free, and is not necessarily able to allow preëmpting.

Priorities can be defined internally, whereupon they uses a measurable quantity. This includes time limits, memory requirements, memory consumption, and so on. It may also be defined externally, usually by the user. In this case they are usually ranked by importance, consider 2 people paying for time on a cloud service, one of whom is paying for higher priority than the other. Priority scheduling has a preëmptive version. However, there are major problems, with infinite blocking, and starvation. We can solve this through ageing, where we gradually increase the priority of processes that wait for a long time.

2.2.6 Round Robin (RR)

A small unit of time, called a quantum, is defined. This is generally in the range of 10-100ms. The ready queue is a circular FIFO queue. The CPU goes around the ready queue, and allocates each process the CPU time of up to 1 time quantum. When the process uses less than 1 quantum, it voluntarily releases the CPU, and when it is longer, then the process is preëmpted.

Assuming the cost of context switching is 0, is RR always better than FCFS? Let us consider having 10 jobs, all starting at 0, and all of them requiring 100 seconds. Let the RR quantum be 1 second. In this case, both FCFS, and RR will finish at the same time, but the average response time is much worse under RR, since a job can run, then be preëmpted, and continue to execute only far in the future (100 quanta later).

The performance of RR will depend on the running time distribution of the jobs. It is poor if the jobs variance is small. Heuristic measurement has shown RR to be good for "real life" jobs. However, context switching hurts the performance a lot. A context switch means running other jobs, which changes the content of the cache, so even assuming a 0 time context switch, the performance can still be bad.

2.2.7 Multilevel Queue Scheduling

Algorithm:

1. Partition the ready queue into several separate groups
2. Each group has its own scheduling algorithm
3. Scheduling is being done among the queues, for example Fixed-Priority preëmptive scheduling, pr time slice between the queues

Processes are classified into different groups, foreground (interactive) processes, and background (batch) processes. This being starvation free / preëmptive is dependent on the implementation.

To resolve potential starvation, we can add feedback. When a process finishes its execution, it gets feedback, and can move to a different queue. Consider using too much CPU time, and being moved to a lower priority queue. Or perhaps a job waits too long, and it can then move to a higher priority queue (like ageing). The multilevel feedback queue scheduler is defined by the number of queues, the scheduling algorithm for each queue, the procedure used to determine what queue a process enters when it needs a service, and the procedures to determine when to upgrade/demote a process.

3 Parallel systems

These are for when not only do I have many jobs to be run, but I also have many processors on which to run them. Supercomputers are extreme examples of these.

3.1 Supercomputers

Supercomputers provide an extremely high speed of calculation. They are usually comprised of many computers connected together, called nodes, with their own OSs, and are connected via an extremely low latency, and high speed internal network. Supercomputers run parallel jobs, each job can use several processors / nodes, and usually run **without** preëmption. Structures such as those that power supercomputers can also be used to power cloud computing.

3.2 Scheduling

In supercomputers, the scheduler is not part of hte OS, and is instead located above the nodes. Its purpose is to determine which processors will be allocated to which jobs. A known scheduler is SLURM. You can configure SLURM with other schedulers, such as FCFS, SJF, and backfilling scheduler.

In the offline settings, we mentioned that SJF achieves the minimal waiting time, so it is easy to solve the problem for **one** processor. Solving for 2 or more processors is NP-hard, however, there are approximation algorithms for this. The problem is, in the real world, these approximation algorithms perform poorly, or are too computationally expensive in comparison to heuristics.

3.3 EASY scheduler

FCFS is not good, because it can delay processes, even when unnecessary. Simple backfilling (seeing that a job that has arrived later can be run before an earlier job, since we have the processor availability for the second, but not the first, so we run the second first), is not good, because it might cause starvation. EASY is the best of both worlds, it is relatively

fair, and uses backfilling based on arrival times.
EASY keeps 2 lists: a list of running jobs, where it is saved on which processors they are running, and their estimated termination time, and a list of waiting / arriving jobs, where it also saves how many processors each job needs, its expected running time, and they are sorted by their arrival time.

The EASY scheduler operates as follows:

1. Schedule jobs on available processors in FCFS order
2. Make a reservation for the first job which cannot run (Due to insufficient processors or the like)
3. Schedule additional jobs provided that they do not conflict with the reservations

Backfilling requires that the backfilled jobs terminate before the reservations, or that the backfill **only** uses processors not required for the reservations.

3.4 Running time estimation

When users submit jobs, they provide: The number of processors to use, and an estimate of the job runtime. Estimates are used to predict when processors will become free for reservation. They are also used to verify that the backfill job will terminate before the reservation. If it does not, then it will be killed.