CPU-Scheduling

The Illusion of Multitasking

Mark Krutzler
Kantonschule Im Lee — 2024/25

Contents

Part I

Forewords?

0.1 Introduction

We all use computers. The modern person couldn't do half of the things that is expected from him without an instance of Windows, MacOs or any other Operating System. We surf the web, write emails, documents and have a video call at the same time. The question is: "How does CPU achieve all of this?" The things listed are only the parts of the OS that we come in contact with on a daily basis. What about all the other things hidden under the hood of a graphics environment? If it comes down to doing one thing than it's fine. The problem arises then we have multiple tasks: The CPU can only do one thing at a time and yet we all multitask on our machines. This Black Box effect of taking multiple jobs and working on them in a proper order is what I will disect in this "Maturarbeit".

The art of scheduling tasks could seem easy for us humans. We ?instinctively? estimate the many aspects of the task:

- Job: Convince our boss to give us a pay rise
  Priority: High
  Time Required: Medium (With a lot of Waiting Time)
  Effort: High (Said "No Way" last week)

- Job: Walk the Dog
  Priority: High
  Time Required: Low
  Effort: Low

- Job: Go to the Hairdresser
  Priority: Low (Went last week as well)
  Time Required: High
  Effort: Low

These prediction are often highly complex and are based on previous experiences. We also include factors like: When was the last time I did it? In addition to all that every human plans differently based on what's important to him/her. Still from just the priority, time required and effort, we would roughly know how to order these things. For example I would walk the dog first, because it is a "request" made by someone else and therefore the response time should be as low as possible. The Hairdresser can fade into the background, due to its low priority. Maybe making an appointment is a good idea though.

A computer does nothing like that. How could it? What he sees is:

- Job: Process 1
  Priority: Needs to be Set
  Time Required: ??? (Waiting Time: ???)
  Effort: ???

What he knows is when the task arrived and how much time he worked on it. With this in mind the let's get to a quick introduction of the metrics, so that we know how to rate the policies that we'll look at.

0.2 Metrics

MAYBE JUST LEAVING THIS OUT AND LOOKING AT THIS DURING A LATER TIME (WHEN THE METRICS ARE MENTIONNED OR LINK AS A FOOTNOTE?) ALSO THIS FOLLOWS THE OSTEP WAY TOO MUCH: We can measure different aspects of a Scheduler Policy. There is however a differentiation between fairness and performance. It is often a tradeoff between the two. For example the concept of everybody getting a same sized piece of cake is fair. However, it would be faster to give more to the fast eaters than to those who like to talk during eating. In this scenario we can't have both fairness and performance. In reality it is really difficult to predict, who'll eat faster.

Definition 2.1: Turnaround Time

The turnaround time is a performance metric. It measures how long a task took to be completed from the time it arrived. It is calculated as follows:

$$T_{Turnaround} = T_{Completion} - T_{Arrival}$$

We'll often look at the Average Turnaround time for a predetermined set of jobs. This helps us to quantify how "efficient" a single task is handeled. In the best case scenario a single jobs turnaround time is equivalent to the runtime of that job. Another really important thing for us is Response Time. We want to move our mouse around. If we type we want the letter to appear in an instant and the Microsoft suite should start without much delay. Of course the startup time of a program depends on many other factors, however if we never even start to work on it, it'll take a good while.

Definition 2.2: Response Time

The response time is a performance metric. It measures the responsiveness of our computer. How much time does it take until the job is run? It is calculated as follows:

$$T_{Response} = T_{First\_Run} - T_{Arrival}$$

Part II

The ABCs of CPU Scheduling

MAYBE HERE THE METRICS PART?

Chapter 1

Basic Algorithms

This chapter mainly focuses on simple policies that can be used to order processes. Usually there are multiple simplifications added to the scenario so that it makes sense. The policies are generally either an abstraction for though experiments or a part of a larger system. Therefore there will be an absense of answers to common questions like: "How do we know how long a process takes until completion?" [1] There are also some example code snippets to each policy. These are just so you can understand the algorithm and they don't really work in a real life scenario. First of all they can't handle interrupts and therefore don't work in a real system. In addition to that Python is a high level language and is absolutely not made to handle low level stuff! Later on we'll switch to C++, because it fits better, however a "real" OS is written mostly in C.

1.1 First In, First Out (FIFO)

Probably the most basic of them all. The "First In, First Our" is just as the name suggests it. Usually we humans know it as first come, first served. This policies runs into problems really quickly if the discard the possibility of all tasks requiring the same amount of run time. A so called "Convoy Effect" takes place. Just imagen how annoying it is, when you're at a store and a family of five cuts into the line in front of you. They have two full carts loaded. You have to wait until they finish. In addition to that they also send back their kid to get an extra bottle of ketchup. Wouldn't it be overall much faster if they let you in front, before the cashier starts scanning the items?

```python
# Simple implementation of FIFO in Python
list = []

# Adding a new process
def add_client(client):
  list.append(client)

# Decide which process is next
def allocate():
  next = list.pop(0)
  use_resource(next) # function that will allow "next" to use the CPU
```

As you can see the implementation is itself pretty easy.

[1] Actually we actually never really know how long something will take

1.2 Shortest Job First (SJF)

Now the situation changes a bit. The infamous family of five wants to go first, however the cashier picks you first after seeing, that you only have a few items. This is the "Shortest Job First" (SJF).

```python
# Simple implementation of SJF in Python

# Define client class and list
class Client:
  def __init__(self, burst): # burst time is the time required for the
     process to finish
    self.burst = burst
    # other stuff that is relevant for the client class

list = []

# Adding a new process
client = Client(burst)
def add_client(client):
  list.append(client)

# Decide which process is next
def allocate():
  list.sort(key=lambda a: a.burst) # sort according to burst
  next = list.pop(0)
  use_resource(next) # function that will allow "next" to use the CPU
```

There are two notable changes in this code snippet, when compared to FIFO. One of them is relevant. First, the less exiting part is that I actually took my time to write out some of the client class definition. Second and the real difference to FIFO is that we sort the according to the burst time[2]. This means that we will choose the task with the shortest time to completion. AT THIS PART THE KNOWLEDGE OF PROPER SORTING ALGORITHMS IS MISSING: MAYBE A REVERSE BUBBLE SORT? THE LIST SHOULD BE ALREADY SOMEWHAT ORDERED AND THE LAST MEMBER HAS TO BE SORTED The only problem is when you arrive late. The scanning of products already started and you'll have to wait. This policy is a so called non-preemptive policy. "Non-preemptive" just means that once a task will run until it finishes.

1.3 Shortest Time-to-Completion First(STCF)

If SJF and FIFO are cousins that STCF is the brother. Here we will make SJF "preemptive", meaning that our cashier can now "save" the state of the previously scanned items. Therefore STCF is also called "Preemptive Shortest Job First (PSJF)". Although the switching costs some time, you'd have to wait anyways until our little boy returns with the ketchup[3] The option of saving a state comes in handy, especially if a program has to wait for I/O. The state of waiting is usually called "blocked" and the technique of running something else during the task is blocked is called overlapping. Preemptiveness also makes the computer much more response, because now you can really do certain things "at the same time". The best part is that we don't even have to change much of the previous code.

[2]Burst time is basically the runtime needed until the process finishes. This variable can be only predicted

[3]If you don't know what I am talking about read Chapter 1.1 (FIFO)

```python
# Simple implementation of STCF in Python

# definition of client class and list go here
current = Client(burst)
current_runtime = 0

# Adding a new process
def add_client(client):
  list.append(client)
  allocate()

# Decide which process is next
def allocate():
  save_state() # saves of the current process

  current.burst -= current_runtime # first update burst
  current_runtime = 0

  list.sort(key=lambda a: a.burst) # sort according to burst
  current = list.pop(0)

  use_resource(current) # function that will allow "next" to use the CPU
      (also tracks runtime)
```

IDK ABOUT THE CODE; IT IS JUST KINDA HACKED TOGETHER; IT IS DEFINITELY NOT SECURE WHATSOEVER! Whenever a new client ”joins”, the ”allocation” function gets run. In addition to that, whenever the process finishes, or it voluntarily gives up the CPU, the same ”allocation” function can be run.

1.4 Round Robin (RR)

Now, why don't we just scan one item for everyone? This would mean that, even though slow but progress is being constantly made. At this point the item scanning doesn't really work anymore[4] but let's just make the so called ”quanta” (quantum in singular) even smaller. As already mentionned in the previous chapter, the cashier needs some time to ”save” the state of the current purchases. This effectively mean that after the quantum shrinks to a certain extent, the ”context switching”[5] would get too expensive. The goal is to reach a balance, where we have enough responsibility, but the price for it is not too high. Notice how by treating the processes more ”fairly” by don't caring about their overall burst time, we get rid of a variable, which can only be predicted and replace it with another one, which is more abstract and global.

[4]You can't scan a fraction of a product, or can you?
[5]Context switch saves the state of a process so that it can be restored later

```python
# Simple implementation of RR in Python

list = []
quantum = 1 # Adjust the time slice here

def add_client(client):
  list.append(client)

def use_resource(client, quanta):
  # definition of use_resource
  # now it also takes an amount of quanta that the process can run for
  # after that it saves the state and returns.

for client in list:
  use_resource(client, quantum)
```

At this point it is time to advance to move onto more advanced policies.

Part III

Advanced Policies

Chapter 2

Multi-Level Feedback Queue

2.1 Abstract Rules

Introduction

Here we'll look at policies doesn't "just invent". These are usually more sophisticated and can be hypothetically used in a real system (at least with some adjustemts). First we'll look at the Multi-Level Feedback Queue, short MLFQ. This is one of the most known ones an is used in Solaris. The creator Fernando J. Corbató even recieved a Turing Award for it in 1990. First I'll write down some abstract rules. These are usually complemented with an example. After that there will be a quick implementation of the policy in C++. Please feel free to jump around to get a better grasp of the (theory).

Basics

The most basic principle of MLFQ are the so called "Priority Queues". These are priorities that a program can belong into. They can change over time and these changes reflect the action/state of a program. The changing of the priority is done as followed: every task has a certain amount of "allotment" for each priority. If this CPU time is used up, than the process will be moved down once and recieve a new set of allotment.

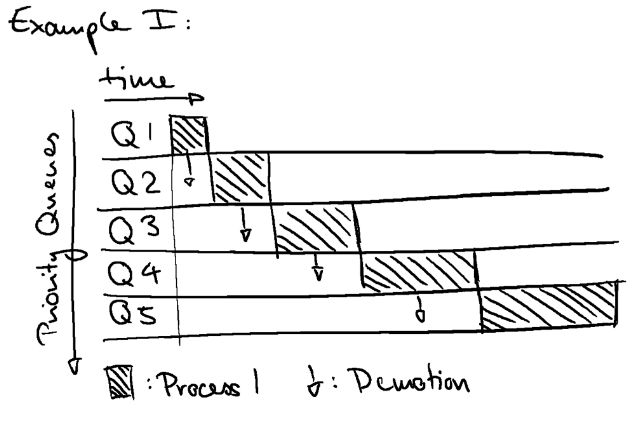

Figure 2.1: Basic Overview

As you can see in the figure 2.1, we have 5 queues. They are represented on the y axis. Although there are usually more than just five, too many queues would just make things more complicated. The x axis represents the time. After a while process 1, gets demoted from the current queue. This happens, because it used up it's allotment. None the less, process one keeps running, because there is no other competing. However if the add another process 2, than the situation changes.

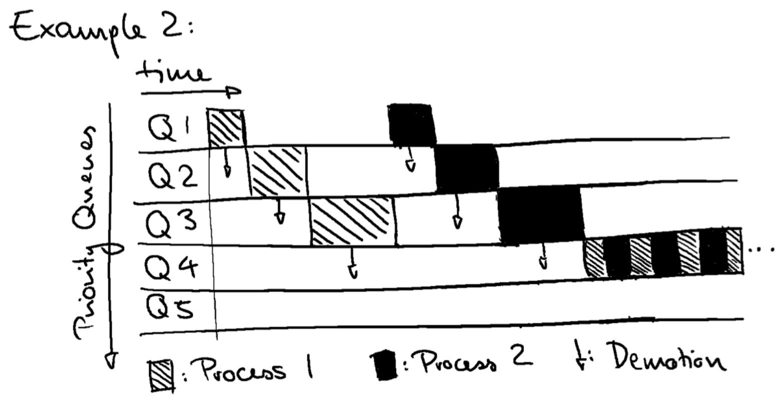

Figure 2.2: Introducing Process 2

As you can see processes at a higher priority are preferred over processes lower down. This effectively means that once process 2 joins the comptetition it can monopolize the CPU until they end up on the same level. Once they are there Round Robin is used to give them both a fair share of allotment. With each process starting in the highest queue, this means that the shorter tasks are generally preferred, because they can finish quickly, before they end up on a lower queue. Therefore the system becomes more responsive. Keep in mind that if a process gives up it's CPU before the allotment is used up (for example if they wait for I/O), than they will "stay" in the same priority and can use up the rest of the CPU time, before getting demoted. Therefore not the actual time spent there matters, but the time spent using the resource. As you might have to already noticed poor process 1 didn't get any runtime while process 2 was getting their priority prioperly adjusted. Note: for simplicity there are only 4 queues.

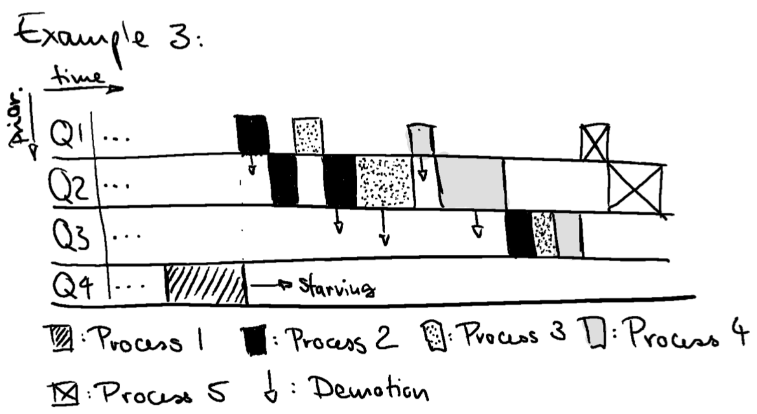

Figure 2.3: Starvation of Process 1

To solve the so called "starvation" issue, which essentially means that longer tasks get to the bottom priority and stay there being stuck, because there is no way to move, we'll introduce "priority boosts". This effectively means that after a certain period S, all processes are boosted to queue 1 and therefore it gives everybody a "second chance". If the processes finish still in priority 1 or 2, than they deserved that queue. If however the processes do not finish than they get promoted together to the bottom queue and everybody recieves a fair share.

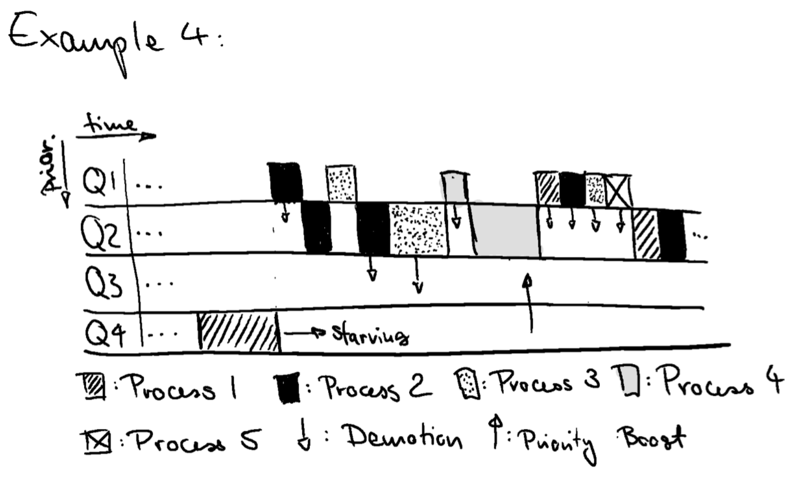

Figure 2.4: Priority Boost

So as "OSTEP REFERENCE HERE" would summarize it:

1. If Priority (A) > Priority (B) ⇒ A runs & B doesn't
2. If Priority (A) = Priority (B) ⇒ A & B run in RR
3. When a job enters the system, it is placed at the highest priority
4. Once a job uses up its time allotment at a given level, its priority is reduced
5. After some period S, move all the jobs in the system to the topmost queue

Voo-Doo Constants

Even though MLFQ introduces multiple Voo-Doo Constant, like the scheduling quantum, amount of queues, period S, allotment size (constant or changing?), we still gain an advantage over just simply unknown constant like the burst time or a predetermined priority and effort. These are huge improvements, because this means that we could actually implement it properly and the use case is not just purely a though experiment. It makes sense to find some sensible defaults for these constants and let the system admin adjust it, if they feel like it. One could find these magic numbers through machine learning, however that is out of scope of the topic?. LINK THE ONE LINK HERE ABOUT ADJUSTING WITH MACHINE LEARNING

2.2 Implementation

Chapter 3

Lottery and Stride Scheduling

Part IV

Real Life Usecases

Chapter 4

Solaris Scheduling

Chapter 5

Linux 2.6 Fair Scheduler

Part V
Conclusion

Sources

- https://texblog.org/2015/09/30/fancy-boxes-for-theorem-lemma-and-proof-with-mdframed/
- https://wiki.osdev.org/Context_Switching
- https://en.wikipedia.org/wiki/Fernando_J._Corbat
- https://github.com/jkoritzinsky/xv6-MLFQ