Kantonsschule Im Lee, Winterthur  
Maturitätsarbeit HS 2024/25

CPU-Scheduling

What waiting in line at the supermarket and your computer have in common

by Mark Krutzler 4a  
under the supervision of Thomas Graf

06. January 2025, Winterthur

The ABCs of CPU scheduling

Initial Problem

In this chapter, I will give a quick introduction to the basics of CPU scheduling. We will look at the proper terms and common approaches to solving the problem of how one should schedule the processes waiting to be run.

Welcome to Costco

The main problem we face can be broken down into a really easy to formulate but hard to answer question. How can we order the queued tasks so that they run in the most optimal order? What optimal really means is a whole discussion itself (not to mention the search for the perfect algorithm to achieve that desired best solution). In this chapter I will give you a quick introduction to the main terms. In addition, I will also try to make the entry into the world of CPU scheduling as easy as possible to understand by using a situation that most of us have already suffered through.

Have you ever wondered why so many people buy bottled water? How they can eat those nasty snacks that you despise? Why they have the whole cart full of milk? Maybe these questions seem rather sudden, but they have two things in common: One, you never get them answered, and two, which is the relevant part, you ask them while standing in line at a supermarket. You ask them while staring at the family of five with two shopping carts and while questioning, whether or not you should have queued at another line. Also, wouldn't it be much better if you, with your two items, went before them? What is the best order to queue up these people? As you can already see, this is the original, simplified question, just reformulated. We are searching for the best algorithm that arranges people at the supermarket. As a matter of fact, let's just call it by its real name: policy. In CPU-Scheduling we call the algorithms policies or disciplines. A single customer / shopper is called a job or a process. In the real world, a process can be anything from your drivers to the web browser showing the cute cat images. A queue is a line of people waiting to get their items scanned and the cashier is the CPU.

How We Compare Policies

Before we dive deeper into the theory, it is important to note that there is a huge difference between fairness and performance. Fairness usually ensures that everyone receives the same amount of CPU time. This kind of policy usually leads to a more responsive system, because each job gets a bit of activity every so often. The response time is defined as:

FORMULA

As already mentioned the other measurable aspect is "performance". Performance is usually measured using the turnaround time.

FORMULA

There is always a tradeoff between performance and responsiveness, because to achieve the best average turnaround time, the tasks can't be interrupted, or else the time of completion is dragged out. Think of the average turnaround time like the predicted time that you have to wait in line until you finish with the payment of your groceries. If you constantly get interrupted then this time will go up. To achieve the best fairness the tasks need to be interrupted, because else new processes have to wait until the previous finishes.

The two formulas above are for a single process. To get a feeling of the overall system, we will look at the average. This average highly depends on the system, because the length of the jobs scales with the average turnaround time proportionally. If on average there are more items for the cashier to scan, then the average time until he is finished will go up. Therefore we’ll have to look at the measurements for a predetermined set of jobs. If we do not have the same system, we cannot compare the policies.

First lines of Code

During the next few chapters and sections we'll assume that one knows the amount of time that a process will take to finish, also known as burst time. Burst time is the time that a process has to run until it is finished. Like bursting out of joy, when you finally finish your weekly shopping. In a real world this is almost impossible(Here I mean knowing the burst time and not escaping the supermarket.), unless you can time travel. One of the most straightforward policy is called first come, first served. This is what we usually suffer through in the queue to pay. The policy disregards how many item one has in the cart. The only important attribute is the time you arrive. The sooner the better. In the world of computer science, the policy is better referred to as First In, First Out (FIFO). Take a look at listing for a basic python implementation.

# FIFO implementation in python

queue = [] # initialize empty queue

# Adding a new process

def add\_process(process):

queue.append(process)

# Schedule the processes

while True:

if new\_process == True: # check if there is a new process

add\_process(process)

next = queue.pop(0) # picks next process

use\_resource(next) # uses resources until finished

Python: First in, First out

We save the queue of people as a python list. If any other job joins, it will just get appended to the end. As for the scheduling itself: We just loop through the list until everyone is finished.

In the example above we assume that the useresource function on line 14 is already written. Also it is important to mention that the newprocess variable is just a bool, which gets updated if a new process is waiting to join the queue. In general python is not really meant for low level programming anyways, so take it with a grain of salt.

Evolving the Supermarket

Now that the scene is set, we return the original question. How can we reduce our waiting time? To give an answer, we will first take a step back. While the previous chapter talked about the initial situation and proper terms, this one will first loosen up some of the rules in order to ensure that the policies are as easy to understand as possible.

Shortest Job First

As the name already says, the shortest job will be run first. In order to achieve this, we assume that the cashier knows, who has the least amount of items. This buyer will be then handled before anybody else. As you can already tell, this is not really a fair way to treat your customers. Still if it comes down to pure turnaround time it is in most cases far better (With the assumption that the tasks arrive at the same time) than the basic First In, First Out. Waiting for shortest jobs to finish first is far less severe for longer jobs, because they already have a long burst time. However making the short processes wait out the long ones increases the turnaround time by a lot. For example let's take process A with a burst time of 10 second and process B with 100 seconds. In case of letting A run first we will have and. Therefore, the average turnaround time is:

FORMULA

In contrast to that if we let process B run first, we will have the turnaround time of 100 and 110. Therefore, the average is:

FORMULA

# Shortest Job First implementation in python

class Process:

def \_\_init\_\_(self, burst):

self.burst = burst

# Adding a new process

def add\_process(process):

queue.append(process)

# Schedule the processes

while queue != []:

if new\_process == True: # check if there is a new process

add\_process(process)

queue.sort(key=lambda a: a.burst) # sort according to burst time

next = queue.pop(0) # picks next process

use\_resource(next) # uses resources until finished

Python: First in, First out

As already mentioned in the section , python is not really meant for writing low level code and therefore it should only be looked at as a simple way of expressing logic. Even though there were some major improvements, the fact remains that if a client is too late then it will have to wait at least until the previous shortest customer finishes. This can take a long time and it would be in our best interest to give these newcomers a fair chance. It would be great if a rescheduling happens, every time someone joins the queue.

Upgrading the Cashier

In this section our cashier receives a significant upgrade: it can save the state of one buyer and switch to another one. In technical terms a policy that can do that is called a preemptive policy. The switching itself is called a content switch. While previously, if another customer came with just a single item, it would not get the privilege of cutting in front of the line. He would only receive the priority boost, if the current person is finished. However, now with the preemptive capabilities the cashier can just save and put aside the people. This feature is not only useful if there is someone that has a shorter time to completion. Imagine that someone forgot to put something into their basket. Now they desperately send back their kid to get that item. While he returns the cashier can save the state and do a context switch. While processes wait for I/O, others can run in their place.

This waiting state is called being blocked and the technique of running another process is overlapping. At this stage it does not really make sense to present some code, because it is just too out of place, however, in theory, one would do the sorting of the processes every time a job finishes, gets blocked or another process joins.

What about Fairness?

All of the policies we looked at so far disregarded the emotions of humans. Most of us want to be treated at least as well as others. It does not matter whether in a supermarket or at home before the computer, we want fairness and responsiveness. A fair scheduler like the Round Robin does exactly that. It is the most simple to implement for a dynamic system, because all it does is goes around and gives every one contestant a bit of CPU time.

The so called quantum or quanta in plural determines how long a single process can run before it gets replaced by another one. In supermarket terms: How many items get scanned per customer per cycle. The logical next step would be to make these quanta as small as possible making the system as fair as possible. This, however, has a heavy drawback.

The context switching itself takes up resources, meaning that after a while the performance degrades, because of too many context switches. Therefore the goal is find a quantum, which balances responsiveness and fairness with the performance loss.

Conclusion

During this chapter we learned the basic methodology of scheduling. The next step is to move to more sophisticated policies, which act as a whole system rather than just as a solution to a specific problem. We will also need to figure out the burst time. To be more precise, we need to figure out how to avoid using it. Most of the above mentioned policies, except the Round Robin, rely on the fact that we know, when one process finishes. Take Shortest Job First for example: How can we order the processes according to the time to finish if we don't even know when they will finish?

Advanced Policies

After looking at the basics, it is time to dive deeper. Next, we will learn about policies that do not rely on the burst time to function. This means that technically they can be used in a real world application. In reality however, they are usually first modified to fit the environment and the needs better. We will get to know two subclasses of schedulers: real-time and proportional-share.

Real-Time Scheduling

The first policy we look at is officially called Multi-Level Feedback Queue, or MLFQ in short. The creator Fernando J. Corbató received a Turning Award for it in 1990. As the title already says, this policy tries to predict the future behavior of the processes based on the past. A job can generally act in two ways: Either it is a resource-intensive crunching problem (think about exporting a video or compiling code) or it is a program, which needs quick response time (think about your text editor). In reality most jobs jump between these two states.

We usually want to give the response-focused processes priority, because that is what the user interacts with and it is here that he primarily feels a delay. The MLFQ is used in the Solaris 2.6 Time-Sharing scheduler. It is part of the time sharing subclass, meaning that it schedules based on an event-driven model, which is usually periodic.

Basic Idea

The policy introduces multiple queues, which each have different priorities. Each process is assigned to a queue. There are, however, not set in stone. Based on the reasons above we want to assume that a new process is responsive, because in the worst case scenario, we just need to demote the non-responsive ones. If however, a process turns out to be interactive, then the user does not feel any lag. Each process can run a certain amount of time (also called allotment time) before it is deemed as unworthy of the current priority. If the allotment is used up, the process gets demoted into a lower queue.

Simple working of MLFQ

As you can see in figure process one gets demoted after a while. The lower the queue is, the longer the allotment time. This is because we hope that all of the responsive-oriented jobs finish before demoting. Once these are filtered out we will only have resource heavy tasks left. These require more time anyways, so the allotment time is stretched out. After a while the process just ends up at the bottom, at which point it runs until it is finished. Keep in mind that if the process gets blocked, meaning that it gives up its CPU before the allotment is used up, then they will stay in the same queue and can use up the rest of the allotment before they get demoted. Therefore not the actual time spent in a priority matters but the time spent using the resource.

Multiple Processes

What happens if we introduce another process? Well, it depends on the priorities. Higher priority receives the CPU time. If they are on the same queue, then they run using Round Robin (see section ).

Running multiple processes in MLFQ

As you can see in figure , once process two is introduced, process one is temporarily starved. The problem is solved once they land on the same queue. There they run alternately. Still the more tasks we introduce the more prevalent the starving issue gets. Take a look at figure for example. Here we have four processes instead of two, which results in process 1 being completely left out. It could only start running again if processes 2 to 4 either finish or get demoted into priority 5. That takes quite a lot of time and requires that we don't introduce new tasks during the time waiting.

Starving Processes in MLFQ

Solving Starvation

Solving starvation is actually pretty easy. All we have to do is to make sure that once in a while everybody gets their deserved CPU time. To do that we introduce a priority boost. All it does is just put every process into priority one every so often. Even though this introduces another unknown variable that has to be configured, it is just such a crucial element that we have to live with it.

Solving Starvation in MLFQ

Wrapping Up

This chapter showed us, how one could approach a scheduling policy that works without needing to know the burst time. In the end we still end up with many parameters like the allotment time per queue, the number of queues, the quanta for the Round Robin and others. If used in a real system, the best approach would be to set some sensible defaults and let the administrator adjust the parameters when needed. To summarize the Multi-Level Feedback Queue, I would like to quote its rules of it from the book OStep:

"If Priority (A) Priority (B) A runs and B doesn't

If Priority (A) Priority (B) A and B run in Round Robin

When a job enters the system, it is placed at the highest priority

Once a job uses up its time allotment at a given level, its priority is reduced

After some period S, move all the jobs in the system to the topmost queue"

Proportional-Share Scheduling

In this chapter we will look at proportional-share scheduling. This type of policy works with weights and assigns them to the processes. For us these weights will take the form of tickets. Tickets are a form of currency for the computer. Think of it like money. These can be handed out by the scheduler and give a process the right to use a resource. Usually, the right to a resource is reflected proportionally by the amount of tickets. Meaning a higher share of tickets will lead to a higher amount of CPU time. Logically, the response time is inversely proportional. The more tickets you have the less time you have to wait until you run. These tickets enable easy comparison between processes. One important difference between money in real life and tickets is that tickets are not consumed when used. Practically this means that if you buy something you won't lose your bill. This property leads to the tickets representing the share of the CPU that the job has. Higher priority processes are weighted more heavily, meaning that they get more tickets.

One could imagine these tickets like reservation places in a restaurant. For now it does not matter how the restaurant assigns the places, like it does not matter how the scheduler policy implements the transition from tickets to CPU time. The only thing that is important is that in the end you will have your places and can eat at the restaurant.

Operating with Tickets

You can do two notable things with tickets. First you can transfer them. This does exactly what it means. You decrease your amount of tickets and increase someone else's. With this you could temporarily boost other processes. Say, for example, you have a server and a client. If the client needs something from the server, then it could temporarily boost the servers resources by transferring some of his own tickets. Of course this requires that the server is trusted or else it could just scam you out of your tickets. The second way to operate with tickets is ticket inflation and deflation. This can only be done by the owner of the currency and it is only something that we will look at the next section.

Ticket Currencies

Ticket currencies are a way for parent processes to manage their child processes. Yes, that's right, not all processes are built the same. A child process is a process that is created by another process. A parent process is a process, which has at least one child process. A job can be both a parent and a child at the same time.

The idea behind ticket currencies is that a parent process creates its own unique currency. Therefore he himself can distribute as many bills as he wants. However more custom bills does not equal to a higher share of CPU time in the whole system. Take figure for example. Even though process A had created 100 tickets and gave 40 to process A in the end it is all about ratios. Therefore the overall runtime that A has is:

FORMULA

There is another hidden benefit. If the parent process receives more global tickets, then if it has a custom currency, there are no further things to do. If, however, it does not have a custom currency than the process has to transfer these new tickets to the child processes. Overall it can get very messy all too quickly.

Example of a Custom Ticket Currency

Another benefit comes with custom currencies: the ticket inflation or deflation operation. One needs a custom currency, because it involves creating or destroying tickets. Unlike the ticket transfer, ticket inflation does not need a sender and receiver. A selected process just gets more tickets assigned. With this the parent process changes the ration of ownership and therefore boosts the selected job.

Lottery Scheduling

Like other things in computer science the name already says a lot. Here randomization is exploited to achieve the proportional share in runtime. The concept is, that even if over a small scale there will be unavoidable errors, the longer the scheduler runs the more accurate it becomes. Think of the tickets as lottery tickets. If the randomization is fair, than the more tickets you have the more you will win. There is a downside though. The randomizer algorithm can get pretty expensive. There is a way to reuse one random seed though, but we will only quickly touch upon it later. For an example implementation in C take a look at listing , which is written by Carl A. Waldspurger.

client\_t current; /\* current resource owner \*/

list\_t list; /\* list of clients competing for resource \*/

int global\_tickets = 0; /\* global ticket sum \*/

/\* initialize client with specified allocation \*/

void client\_init(client\_t c, int tickets)

c->tickets = tickets; /\* initialize client state \*/

global\_tickets += tickets; /\* update global sum \*/

list\_insert(list, c); /\* join competition for resource \*/

void allocate()

int winner, sum;

client\_t c;

winner = fast\_random()

/\* search list to find client with winning ticket \*/

sum = 0;

for (c = list\_first(list);

c != NULL;

c = list\_next(list, c))

/\* update running sum, stop at winner \*/

sum += c->tickets;

if (sum > winner)

break;

current = c;

use\_resource(current); /\* grant resource to winner for quantum \*/

Implementation of List-Based Lottery Scheduling Algorithm

Written by: Carl A. Waldspurger

In general implementing dynamic operations is not hard, because the only part that changes is the scheduling part, like choosing a winning ticket. The clients structure does not need to be modified. This is because there are no global states. The processes are kept completely isolated from each other and just by looking at the next scheduling you couldn't know how long the scheduler has been running. In a List-Based system adding and removing clients can be as easy as adding them to the list and adjusting the global max tickets.

Binary Search

Even though it is easy to implement, basing the data structure on a list is quite inefficient, because of the long search time. Every time you want to reschedule, you have to create a random number and loop through the list. However you could use a balanced binary tree to make it much faster. This is because a list search time scales with n. Therefore if you add one more element you will have to search through one more element. In contrast a binary search scales not with n. To be more precise it takes to find a specific value. On average, with each decision we can eliminate half of the remaining candidates. Therefore overall it is much faster, especially the more values the list contains.

Binary Search Example

To understand it more, let's go through an example. Let's say there are 48 tickets and our winning one is 20. To understand the though process better, feel free to look at figure. For our first decision we need to choose if 20 is above 10 or not. It is above ten, so we choose the right path. At this point we have only 38 tickets left and we have to readjust our winning ticket. We took away 10 tickets from the front so our new winning tickets is 10. 10 is smaller than 20, so we take a left turn and there is no readjusting needed. In the next step we compare 12 and 10. 12 is larger, which means we take the right path and arrive at our winner process, because 10 is contained in 12.

Multi-Winner Lottery

Multi-winner Lottery Scheduling is a semi-randomized, semi-deterministic scheduling method. It is a method to try to minimize the cost of the randomization algorithm. The basic idea is to reuse the winner. After you select your winner you take a specific offset to determine the other winners. One of these lotteries is called a super-quantum.

Stride Scheduling

Another approach to implement a proportional-share policy is the so called stride scheduling. The tickets are used to assign so called strides to the processes. This stride value will then be used to increment a counter called pass. The process with the lowest pass value than gets run.

Stride Scheduling Example

To understand the policy better, let's loot at the example in figure. We have three processes. Process A has a stride of 160, Process B 320 and Process C has 30. At process A runs. Afterwards it's B's turn. C has a much smaller stride, that's why it will have more turns before it catches up to A. After a while it gets to cyclic pattern.

The benefit of stride scheduling is that one doesn't need a randomizer and the policy itself is much simpler to implement. With such good benefits there comes some disadvantages as well: The processes have global values, meaning that if a new job wants to join, than it has to be adjusted to the current global circumstances. In contrast all you have to do in the lottery is add the assigned tickets to the global maximum tickets.

To solve this, one needs to introduce a constant global pass value, because otherwise the joining processes will monopolize the CPU. Look at figure for example. If a process D joins at the last time interval, then it has to catch up to at least C. During that time it will have a unreasonable priority over everybody else. A promised global pass value would eliminate the problem, because one can adjust the entry pass value according to the current situation. If a task goes into a blocked state or leaves the competition temporarily and than rejoins than its pass value will be global stride plus the stride of the process. The same goes for a brand new job that joins.

Remzi Hussein Arpaci-Dusseau Andrea Carol Arpaci-Dusseau (2018) Operating Systems: Three Easy Pieces, CreateSpace Independent Publishing Platform.

Edward G. Coffman Leonard Kleinrock (1968) Computer scheduling methods and their countermeasures, Association for Computing Machinery.

Carl A. Waldspurger (1995) Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management, Massachusetts Institute of Technology.

Carl A. Waldspurger William E. Weihl (1994) Lottery scheduling: Flexible proportional-share resource management, Massachusetts Institute of Technology.

Andrea Carol Arpaci-Dusseau (2000) Multilevel Feedback Queue Scheduling in Solaris, available at: [https://pages.cs.wisc.edu/ remzi/OSTEP/Citations/notes-solaris.pdf](https://pages.cs.wisc.edu/%20remzi/OSTEP/Citations/notes-solaris.pdf)

Further Sources

<https://github.com/Pseudomanifold/latex-mimosis>

<https://en.wikipedia.org/wiki/Binary_search>

<https://www.quora.com/Why-is-look-up-faster-for-a-Binary-Tree-than-a-Linked-List>

<https://ceunican.github.io/aos/09.Scheduling_Proportional_Share.pdf>

<https://courses.cs.washington.edu/courses/cse451/12au/l11.pdf>

<https://en.wikipedia.org/wiki/Earliest_eligible_virtual_deadline_first_scheduling>

<https://en.wikipedia.org/wiki/Proportional_share_scheduling>