

Elements of Operating Systems

Chapter 6

Scheduling

J. Fornés M. Farreras Y. Becerra

Barcelona School of Informatics
Universitat Politècnica de Catalunya

Bachelor's Degree in Informatics Engineering — Operating Systems

Preview for readers · September 28, 2026



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

© Universitat Politècnica de Catalunya

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International licence (CC BY-NC-SA 4.0):

«By Universitat Politècnica de Catalunya — BarcelonaTech (UPC), Any 2026»



<https://creativecommons.org/licenses/by-nc-sa/4.0/deed.en>

This text was written with the help of Claude, Anthropic's language model. Any errors that remain are the authors'.

6 Scheduling

In Chapter 5 we kept leaving things for later. That there is a queue of ready processes and that `fork()` adds the child to it; that a process goes from RUN to READY without asking; that there is a clock interrupt that lets nobody monopolise the machine. All of that is this chapter.

Let's say at the outset what we are after. There was a time when scheduling was the heart of an operating system, because the CPU was the machine's scarce and expensive resource. Today there is often CPU to spare and what is short is something else — memory, bandwidth, battery —, and real schedulers have grown complicated enough that studying them line by line does not pay. What does pay, and handsomely, are the two or three ideas underneath, because they turn up everywhere: what it costs to change process, why there has to be a time limit, and what it means for a share-out to be fair.

6.1 Policy and mechanism

The distinction that orders the whole chapter is this one, and it is one of the great ideas of operating systems [1].

Policy decides *what* has to be done: whose turn the CPU is now, and for how long. **Mechanism** is *how* that decision is carried out: how one process's state is saved and another's restored. They are kept apart on purpose. The mechanism is expensive to build and almost never changes; the policy can be changed — and Linux has in fact changed it several times — without touching a single line of the mechanism.

Policy belongs to the *scheduler*. Mechanism, to the context switch.

6.2 What really gets shared out

Before looking at where the kernel keeps its processes, we have to be precise about what it is sharing out, because the answer that comes to mind is a half-truth.

The answer that comes to mind is “processes”. In Chapter 12 we will have to correct it a little — what the scheduler picks is a thread, and a process with a single thread is both things at once —, but the correction that counts is a different one, and it is this.

Look closely at the life of a process. It is not one block of work lasting from `fork()` to `exit()`: it is an alternation. It computes for a while, asks a device for something and waits; when it has it, it computes for a while again, and asks again. Each stretch of uninterrupted computation has a name: it is a *CPU burst*.

And a burst does not end for no reason. It ends in one of two ways, which are exactly the two of Chapter 9: either the process makes a call that blocks it — a `read()` from a terminal, a `write()` to a full pipe — and gives up the CPU itself, or it does not end and the CPU has to be taken away from it.

The scheduler does not share out processes, nor even threads: it shares out **CPU bursts**. Every time it runs, the question it answers is which burst goes now; which process it belongs to is a consequence, not the question.

This is not a scholar’s quibble: it changes the meaning of everything that comes after.

- The ready queue is not “the processes there are”, but those that have a burst started or about to start. A blocked process is not in it because at that moment it has none.
- The quantum is not “how long a process lasts”: it is how far we let a burst get before cutting it short.
- And when we compare two policies, what we compare is how they order bursts.

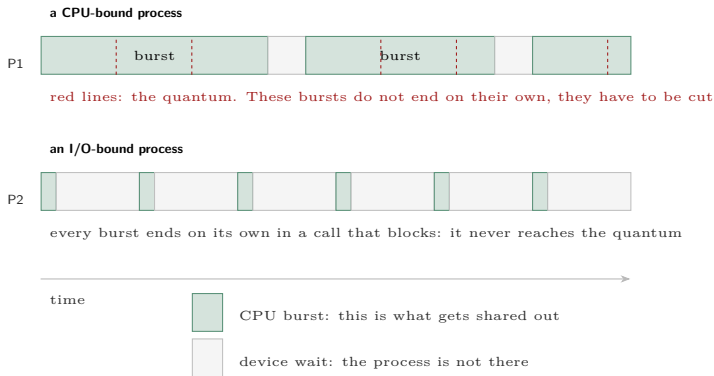


Figure 6.1. The same two processes on the same time scale.

6.2.1 Two ways of living

Almost every process alternates bursts and waits, but not in the same proportion, and the difference is not one of nature but of size:

- A **CPU-bound** process makes few bursts, and very long ones. Often it makes only one, which lasts the whole program, and which the quantum keeps cutting into pieces. A compilation, a video being compressed.
- An **I/O-bound** process makes a great many, and tiny ones. A text editor: between one keystroke and the next it does nothing, and when you press one it computes for a few microseconds and goes back to waiting.

i Careful

“CPU-bound” and “I/O-bound” are not labels the process carries stuck to it: they are descriptions of what its bursts look like, and they can change while it runs. The same compiler is an I/O-bound process while it reads the files and a CPU-bound process while it optimises. The scheduler does not know in advance, and does not need to: it only has to look at what has happened so far.

6.2.2 📌 Looking at the bursts

All of this can be checked without going into the kernel, because the kernel keeps the count and lets you look at it: how many times a process has left the CPU because it wanted to, and how many times it has been taken away. The sum is how many turns of CPU it has had.

Listing 6.1. sched-bursts: how many turns, and how big

```

1  int main(int argc, char *argv[])
2  {
3      double cpu, turns;
4      struct rusage r;
5      char buf[192];
6      long n;
7
8      if (argc != 3 || (n = atol(argv[2])) < 1)
9          Usage(argv[0]);
10     if (strcmp(argv[1], "compute") == 0)
11         compute(n);
12     else if (strcmp(argv[1], "io") == 0)
13         io_wait(n);
14     else
15         Usage(argv[0]);
16
17     getrusage(RUSAGE_SELF, &r);
18     cpu = r.ru_utime.tv_sec + r.ru_utime.tv_usec / 1e6
19         + r.ru_stime.tv_sec + r.ru_stime.tv_usec / 1e6;
20     turns = r.ru_nvcsw + r.runivcsw;
21
22     sprintf(buf,
23             "%s: %.3f s of CPU\n"
24             "  it leaves the CPU %ld times, it is taken away ↵
25             %ld\n"
26             "  CPU per turn: %.3f ms\n",
27             argv[1], cpu, r.ru_nvcsw, r.runivcsw,
28             turns ? cpu * 1000 / turns : 0);
29     write(1, buf, strlen(buf));
30     return 0;
31 }
```

❶ `getrusage()` gives you the accounting the kernel already keeps for this

process; there is nothing to instrument.

- ② `ru_nvcsw` is the *voluntary* context switches — it gave up the CPU itself — and `ru_nivcsw` the *involuntary* ones — it was taken away.
- ③ And the division: how much CPU it got each time it had it.

```
$ ./sched-bursts compute 2
compute: 2.002 s of CPU
  it leaves the CPU 0 times, it is taken away 3
  CPU per turn: 667.273 ms
$ ./sched-bursts io 2000
io: 0.019 s of CPU
  it leaves the CPU 2001 times, it is taken away 0
  CPU per turn: 0.010 ms
```

Look at the two middle columns before you look at the times. The CPU-bound process **does not leave the CPU once**: it never blocks, and the three times it loses it are because it is taken away. The I/O-bound one is the exact opposite: it leaves it two thousand times and never has to be forced off, because every burst ends by itself long before the quantum does.

And now the times: 667 milliseconds against 0.010. On another lab machine it comes out as eight turns and 250 milliseconds: how many you get depends on the machine and on who else is running. The ratio, on the other hand, hardly moves. Almost five orders of magnitude, between two programs running on the same machine in the same second.

Careful

The 667 milliseconds of the CPU-bound process are not the size of its burst: its burst lasts the whole two seconds. They are the size of the *piece* the system lets it run without a break, and on an empty machine like this one it lets it run very long ones. Try it with three copies at once and watch the figure drop.

The design consequence is the one that explains the whole chapter: in a real system, the vast majority of bursts are short. That is why

a quantum of tens of milliseconds is almost never used up, and why preemption — which looked like the heart of the whole business — turns out to be the rare case.

6.3 The kernel's queues

You know enough about the PCB: in Chapter 5 we saw what is inside it. What is missing is how they are organised, because a thousand PCBs in a heap are no use at all: the scheduler has to be able to answer the question “who is next” in microseconds.

The usual answer is queues. There is a list with every process in the system, and then there is one for each reason a process may be waiting: the **ready queue** with those that want CPU*, and a **queue per device** with those waiting for the disk, the network or the terminal to answer them.

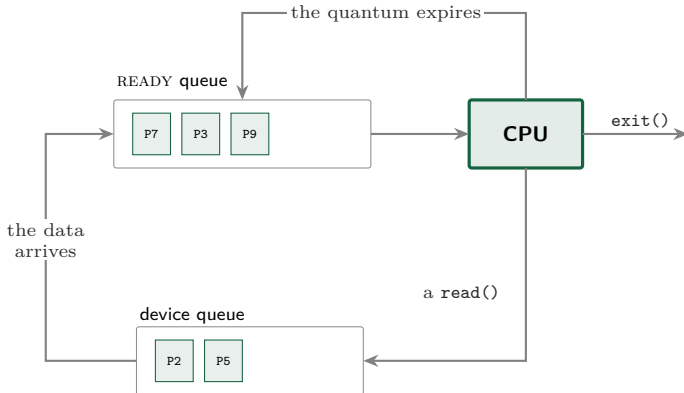


Figure 6.2. A process does not pass through the scheduler: it goes round and round it. Every arrow in this drawing is an arrow of the state graph of Chapter 5.

Comparing this figure with the state graph on page 74 shows that they are the same drawing twice: there, the states of a process; here,

*In practice it is not a single one. Real systems split it by priority, by class of process or by CPU, and on Linux it is not even a queue: it is a tree. We will see this at the end.

the structures where the kernel keeps the processes that are in them. A process is not “in the READY state” in some mysterious field: it is in the READY queue, and changing its state is taking it out of one list and putting it in another.

More depends on the choice of structure than it seems. Inserting and removing have to be fast, because it is done at every context switch; searching has to be fast, and you have to decide which field you search by — by PID? by user?; and the whole thing has to hold up when the machine has not ten processes but ten thousand.

6.4 The context switch

When one process leaves the CPU and another comes in, the kernel has to save everything that lets the first carry on later, and restore the same for the second. That is a **context switch**.

What is saved is the PCB’s hardware context: the registers, the program counter, the state of the floating-point unit, whatever it takes for the process not to notice a thing when it comes back. And the place it is saved in is the PCB itself, which is why it has a hole of exactly the right size.

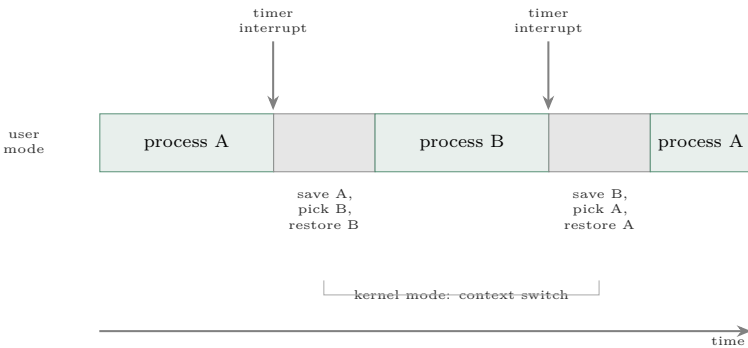


Figure 6.3. The grey pieces are the context switch. They make no program advance: that is why they have to be short.

A context switch is not useful time. While it lasts, no user program advances a single instruction: the machine is busy remembering where we were. Everything this chapter discusses about quanta and frequencies comes, in the end, from wanting that stretch to be a small fraction of the rest.

That is why processors help. Many architectures have instructions that save or restore a whole block of registers at once, and some systems avoid saving the floating-point state until the new process really touches it — if it never touches it, the work was never done.

6.5 When the scheduler runs

The scheduler is not running all the time: it is kernel code, and the kernel only runs when something calls it. The occasions are the two ways of ending a burst that we have already seen, and now they have names.

The process holding the CPU cannot carry on. It has called `exit()`, or it has blocked in a `read()` or a `waitpid()`. There is nothing to decide here about taking the CPU away from it — it has already given it up —, only which one goes next. These events are called **non-preemptive**.

The process holding the CPU could carry on, but it is taken away from it. Its turn is over, or a process the system considers more urgent has just appeared. These are the **preemptive** ones, and they are the ones that require the kernel to be able to take control back from a process that had no intention of returning it.

A policy that only handles events of the first kind is **non-preemptive**: it never takes the CPU away from anyone, and waits for the process to release it. One that handles the second kind as well is **preemptive**, and it needs a clock interrupt, no two ways about it.

That interrupt is what makes the whole thing possible, and it is the answer to the question we left you with in Chapter 5: every few milliseconds the hardware interrupts whatever is running and hands control to the kernel, which takes the chance to see whether the process has to be changed. Without it, an infinite loop in a user program would leave the machine hung forever — which is exactly what happened on the systems that did not have one.

6.6 What we want to optimise

Before comparing policies we have to know what is measured, because otherwise “better” means nothing. The usual measures are these:

Turnaround time the total time from the process arriving in the system until it finishes. Everything counts: the CPU it has used, the time it has waited in the READY queue and the time it has waited for the disk.

Waiting time only the time spent in the READY queue. It is the part the scheduler can improve; the others do not depend on it.

Response time the time from asking it for something until you see the first sign of it. It is what the person in front of the screen notices, and it often matters more than the other two.

Throughput how many processes the system finishes per unit of time.

Careful

No policy optimises them all at once, and choosing one means giving something up. A system that wants the best response time will change process very often, and every change is lost time, so throughput will go down. When you compare two policies, the first question is not which is better: it is better *for what*.

And there is a second thing to keep in mind, which you already know: that the bursts of CPU-bound processes and those of I/O-bound

ones differ by four orders of magnitude. It comes back two sections from now, because that is exactly where Round Robin trips up.

6.7 Round Robin

It is the simplest policy that works, and that is why we explain it even though no current system uses it as it stands.

There is a single rule: ready processes queue up in order of arrival, and each one gets the CPU for a fixed maximum time, the **quantum**. If it uses it up, the CPU is taken away and it goes to the back of the queue; if it blocks or finishes first, it leaves and the next one comes in.

The three events that set it going are, then, the chapter's: the process finishes and the process blocks — non-preemptive — and the quantum runs out, which is preemptive. Round Robin is therefore a preemptive policy, and it needs the clock.

Let's see it with three processes that arrive at the same time and ask for 5, 3 and 4 units of CPU, with a quantum of 2:

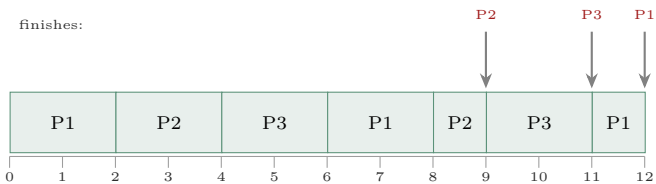


Figure 6.4. Round Robin with a quantum of 2. Nobody runs more than two units in a row while somebody is waiting.

	Asks for	Finishes at	Has waited
P1	5	12	7
P2	3	9	6
P3	4	11	7
Average		10.7	6.7

Notice that P2, which was the shortest, is the one that finishes first, and that this is no decision of the policy's: it comes out on its own. And notice too that all three have waited a long time in relation to

what they were asking for. Round Robin is not good at turnaround time; what it is, is *predictable*.

6.7.1 The quantum

The whole policy hangs on a single number, and the two easy answers are both of them bad.

If the quantum is very **large**, nobody ever uses it up: every process will have the CPU until it blocks or finishes, and Round Robin turns into “first come, first served”. A long process leaves the rest of the machine waiting.

If it is very **small**, processes alternate very fast and the response time is superb, but between one turn and the next there is a context switch. With a quantum the same size as the context switch, half the machine goes on remembering where we were.

The quantum has to be **large compared with the context switch** and **small compared with what people notice**. Between those two marks there are several orders of magnitude, which is why the usual values — between 10 and 100 milliseconds — work well enough without any tuning.

With N processes in the queue and a quantum Q , each process gets an N th of the CPU time, in pieces of at most Q , and none waits more than $(N - 1)Q$ to have some again. That last one is the property that keeps Round Robin being taught: it gives a guarantee, and a guarantee you can work out.

6.8 What Round Robin gets wrong

Go back to Figure 6.1 and put the two strips in the same queue.

The CPU-bound process has a burst longer than the quantum, so it uses the whole of it up every time its turn comes. The I/O-bound one has bursts of microseconds: it does a few things, blocks waiting for a keystroke or the disk, and hands the CPU back after a thousandth of

what it had been granted. Now look at what happens when the two of them queue together.

i Careful

Round Robin gives everyone the same *turn*, not the same *CPU*. The CPU-bound process takes the whole quantum every time round; the I/O-bound one, a crumb. After a second, the one that has kept almost all of it is the one that already had enough, and the text editor moves in jerks.

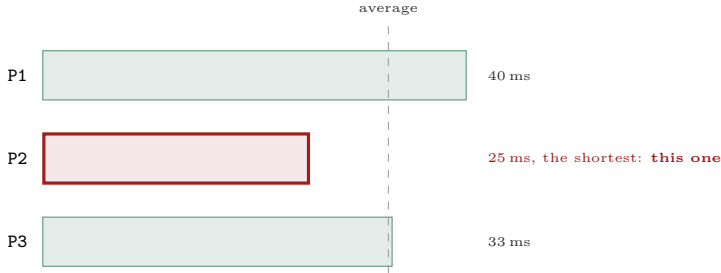
The problem is not that the policy is unfair by oversight: it is that it does something different from what we thought. Sharing out turns and sharing out time are different things, and what we want is the second.

6.9 Completely Fair Scheduling

This is the idea Linux has used since 2007*, and the twist is small and radical at once: instead of sharing out turns, keep count of the CPU time each one has used.

For every process, CFS asks how much CPU it *ought* to have if the share-out were perfectly fair, and how much it really has. It always runs the one that is furthest behind its ideal.

*Since kernel 2.6.23, and under the name completely fair scheduler. Today's kernels no longer carry it: see the box at the end of the section.



Each bar is the CPU that process has used so far. Neither turns nor priorities come into it: the shortest one is always picked, and so the three tend to even out.

Figure 6.5. The CFS decision: the shortest bar.

A number of consequences follow from this.

There is no fixed quantum. The maximum time a process can spend on the CPU without a break is worked out: it depends on how many are competing and on how far each one is from its ideal. With few processes the pieces are long and there is little switching; with many, they get shorter.

I/O-bound processes stop coming off badly, and not because they are treated well: because the counter tells the truth. A text editor that has used two milliseconds of CPU in a whole second is a long way behind its ideal, and so, when you press a key, it is first on the list. The instant response you notice is not a special priority: it is arithmetic.

The queue stops being a queue. If what has to be done is “pick the one with the smallest counter”, a list in order of arrival is no use. CFS uses a balanced tree ordered by that counter, so that picking the next one is going to the leftmost element.

And it can count by groups. The administrator can say that all the processes of one user, or of one service, count as a single one when the sharing out is done. That way, a user who launches a hundred processes does not take a hundred shares of the machine: they take one, and share it among their own.

⚠ Question 1

With Round Robin, opening a hundred windows of a program gives me a hundred times more CPU than my neighbour who has opened only one. With CFS by groups, it doesn't. Which of the two things is "fair"?

💡 Answer

Both, and that is the point. The first is fair between *processes*; the second, between *users*. No policy can decide on its own which of the two fairnesses we want: it is a decision for whoever administers the machine, and that is why CFS leaves it configurable instead of choosing it itself. When you read that a scheduler is "fair", the useful question is always the same: fair between whom and whom.

🔗 Going further

Since kernel 6.6, from 2023, the Linux scheduler is no longer CFS but EEVDF (*Earliest Eligible Virtual Deadline First*), and the lab machines use it. What changes is not the underlying idea —keeping count of the time used and favouring whoever is behind—, but how the tie is broken: each process is given a virtual deadline, worked out from the time it ought to get, and the one whose deadline is nearest runs. With that, the stretch an interactive process may end up waiting is bounded, which CFS promised nowhere. Everything we have explained in this section still holds; the small print is in `Documentation/scheduler/` of the kernel source.

6.10 Looking at it from outside

None of this can be seen from a program, but its effects can. This one spins for three seconds of wall clock and then says how much CPU it was really given:

Listing 6.2. How much CPU did I get

```

1  int main(void)
2  {
3      char buf[96];
4      time_t start;
5      clock_t cpu;
6      double used;
7      long laps = 0;
8
9      start = time(NULL);
10     cpu = clock();
11
12     while (time(NULL) - start < SECONDS)
13         laps++;
14
15     used = (double)(clock() - cpu) / CLOCKS_PER_SEC;
16     sprintf(buf, "pid %d: %ld laps, %.2f s of CPU\n",
17             (int)getpid(), laps, used);
18     write(1, buf, strlen(buf));
19     return 0;
20 }

```

- ❶ `time()` gives wall-clock time; `clock()`, the CPU time used by this process. That they are two different things is the subject of the chapter.
- ❷ The loop does nothing useful on purpose: all it wants is the CPU.
- ❸ And the difference between the two is what tells you how much you have had to share.

Run it on its own and then a few copies at the same time. Watch out for one thing: if the machine has several cores, four copies will really run in parallel and will share nothing. To see the sharing out you have to force them to queue on a single core, and that is what `taskset` does:

```

alumni@labS0:~/S3$ ./sched-share
pid 8169: 1399805598 laps, 2.56 s of CPU
alumni@labS0:~/S3$ for i in 1 2 3 4; do
>   taskset -c 0 ./sched-share &
> done; wait

```

```
pid 8172: 398232609 laps, 0.75 s of CPU
pid 8175: 397429357 laps, 0.75 s of CPU
pid 8174: 400544049 laps, 0.75 s of CPU
pid 8173: 398865396 laps, 0.75 s of CPU
```

Three seconds of wall clock in both cases. On its own, the process takes almost all the CPU of that core; with four competing, each one takes exactly a quarter, with a regularity that is faintly unnerving.

i Careful

`taskset`'s `-c 0` does not mean “core zero”: it means “CPU number zero”, and a **Linux CPU is not a core**. If your machine has SMT —two sets of registers per core, which Intel calls *hyper-threading*—, `lscpu` will tell you that you have eight CPUs when in fact you have four cores, and two of those eight will share hardware without the number saying so.

Worse still: **which two of them share the same core depends on the machine**, and you cannot make it up. You have to ask. This is one of the lab machines:

```
$ lscpu -e=CPU,CORE,SOCKET
CPU CORE SOCKET
0      0      0
1      1      0
2      2      0
3      3      0
4      4      0
5      5      0
6      0      0
7      1      0
8      2      0
9      3      0
10     4      0
11     5      0
```

Twelve CPUs and six cores: the first six are one set of registers from each core, and the next six, the other set. CPU 0 and CPU 6 are the same core, 1 and 7 too, and so on to the end. So in the lab a `taskset -c 0,1` puts the two copies on different cores, and a `taskset -c 0,6` puts them on the same one—and the measurement you took would not mean what you think. And it is no rule: in other rooms of the same faculty CPU 0's sibling is CPU 1. Look at the table before you pin anything down.

And one more: on many of today's machines **the cores are not all alike**—some are fast and some are frugal—, so pinning one copy to zero and another to four can give different times with no mystery behind it. Chapter 12 has how to look at this.

That is the scheduler doing its job, and it is all you will see of it from outside.

Going further

The share-out can be influenced from outside, and the dial is called *nice*: a number from -20 to 19 saying how nice you are prepared to be. The higher it is, the less weight in the share-out and the more CPU for the others. Anyone can raise it; only `root` can lower it, and the reason is obvious.

```
$ strace -qq -e nice,setpriority nice -n 19 ./sched-share
setpriority(PRIO_PROCESS, 0, 19)      = 0
pid 8342: 1660623637 laps, 2.99 s of CPU
```

Look at the name on the first line, which is not the one you typed. The command is called `nice`, the library function is called `nice()`, and what reaches the kernel is called `setpriority()`. In fact `nice()` does not exist as a call on either of this book's two machines: `libc` puts on the show. Appendix A explains why this happens and how to check it on yours.

6.11 Exercises

1. Redo the trace of Figure 6.4 with a quantum of 1 and with a quantum of 5. How do the average turnaround time and the number of context switches change? Which of the two quanta would you choose, and for what sort of machine?
2. Apply a policy that is not preemptive to those same three processes: each one has the CPU until it finishes. Does the average turnaround time come out better or worse? And if P1 had asked for 50 instead of 5?
3. Run `sched-share` with eight copies on a single core. How much CPU does each one get? And if you put a `nice -n 19` in front of one of the eight?
4. Compare the CPU time `sched-share` uses on its own with the time it uses while a compilation is running on the same machine. Is your program CPU-bound or I/O-bound? And the compilation?
5. With `cat /proc/self/status` you can see your own PCB, or the part of it the kernel lets you see. Look in it for the fields `voluntary_ctxt_switches` and `nonvoluntary_ctxt_switches`: they are the two numbers that `sched-bursts` asks `getrusage()` for. Which of the two corresponds to preemptive events?
6. Run three copies of `sched-bursts compute 5` at the same time and compare the CPU per turn with that of a single copy. Now do it with `taskset -c 0`, which puts them all on one core. Which of the two accounting numbers changes, and why?

Bibliography

- [1] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. 10th ed. Hoboken, NJ: Wiley, 2018. ISBN: 978-1-118-06333-0.