CS 194-24 Lab 3: Scheduling

Palmer Dabbelt, Vedant Kumar

March 31, 2014

Contents

For this assignment you will implement a simple real-time scheduler for Linux. It's important to note that we will be using your HTTP server to launch real-time processes. In order to attempt to break the dependency chain as much as possible, none of the performance features implemented for the previous assignment will be necessary here. To successfully test your scheduler you'll just need to be able to launch CGI scripts in parallel. If any group feels their implementation from the previous lab isn't up to stuff then please come and talk with me early and we'll try to work something out.

1 Real-time Scheduling

The meat of this lab consists of adding a real-time scheduler to Linux. You'll be taking advantage of Linux's support for scheduler classes in order to cleanly implement your new scheduler.

For this assignment you will be implementing an extension to the EDF algorithm that allows you to mix real-time tasks with what the paper refers to as "constant bandwidth servers", which are simply processes that get a fixed amount of CPU bandwidth. The CBS paper available on the course website, you should read it before proceeding further

`http://www.cs.berkeley.edu/~kubitron/courses/cs194-24-S13/hand-outs/rtss98-cbs.pdf`

We will be taking advantage of the fact that the CBS algorithm has a natural method of determining the amount of slack present in the system. The paper uses this to criteria to determine if a given set of tasks is schedulable, but the math works out such that we can also determine the largest allocation that we can give CFS in order to maintain the guarantees of CBS.

There are two interesting edge cases for CBS that aren't mentioned anywhere else, so I'm just going to throw them here...

- CBS kills real-time threads by passing `SIGXCPU` to them when they've exceeded their processing time limit.
- CBS must yield the remainder of its time to CFS.

1.1 Linux Scheduler Classes

The simplest way to add CBS support to Linux is to implement a new scheduler class. It's important to note that Linux scheduler classes aren't loadable at runtime (there's no technical reasons this can't happen, it's just that nobody has bothered to implement it). This means you'll need to decide upon your scheduler hierarchy at compile time, which isn't actually a problem because CBS has an clean way to assign the slack to other schedulers.

Linux defines scheduler classes as a simple integer value that defines the scheduling algorithm that decides when to run a given process. Schedulers in Linux have a fixed priority: the highest priority scheduler gets to run all its jobs to completion before the next highest priority scheduler gets to run any jobs. Your new CBS scheduler should be the highest priority scheduler in the system, producing the ordering of schedulers shown in Figure 1.

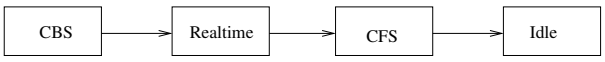

Figure 1: Scheduler Priority Order

In order to make this happen you'll first need to do a bit of build work such that you can install your scheduler beside CFS – reference Lab 0 for how to do this. You'll then need to add a scheduler identifier along side the other identifiers inside `linux/include/uapi/linux/sched.h`. You will need to modify `struct task_struct` in order to keep track of the necessary accounting data you'll be using to

implement CBS. Additionally, you'll need to define a CBS-specific run queue – look at `struct cfs_rq` inside `linux/kernel/sched/`.

In order to actually get your code to be run, you'll need to add some code `sched_init` that initializes your run queue. At this point you can create a new scheduler class that contains a set of function pointers that actually consist of your implementation of CFS (look at the other `struct sched_class` entries in `linux/kernel/sched/sched.h`). Be sure to set your scheduler as the highest priority scheduler inside `linux/kernel/sched/sched.h` with the `sched_class_highest` macro.

At this point you're effectively done with the overhead involved in teaching Linux how to recognize your scheduler.

1.2 Kernel Snapshotting

As you probably discovered during Lab 0, testing realtime processes is inherently difficult because any code you use for instrumentation will inherently break your timings. In order minimize this breakage, we've attempted to design a general but efficient API that allows you to interrogate the scheduler without too much overhead.

We've chosen to use snapshots in order to achieve this. The snapshoting algorithm is defined in `realtime/snapshot.h`. The idea is that you pass an array of (event, trigger) pairs into the kernel. The data is passed into the kernel using the `snapshot()` system call.

When the user calls `snapshot()` the kernel will first empty all of its snapshot buffers. Every time the kernel sees an event that causes the trigger to fire, it fills up the next snapshot buffer with the relevant information. This will hopefully be a very fast operation, as all the buffers will be statically allocated. The user can then go ahead and poll these buffers at their leisure without breaking the hard timing path in the kernel.

This is probably best worked through as an example. The idea is that a scheduling policy looks at the current state of runnable processes and determines which one should be running. A correct scheduler will always make the correct decision as to who to schedule next, so if we sample the decisions a scheduler makes and they're all correct then we can have a reasonable certainty that the scheduler is correct. An example of what the `snapshot()` system call produces can be seen in Figure 2. In this case this would be an invalid run of the CBS scheduler because the wrong process was chosen to run next – it should be B, as B has the earliest deadline.

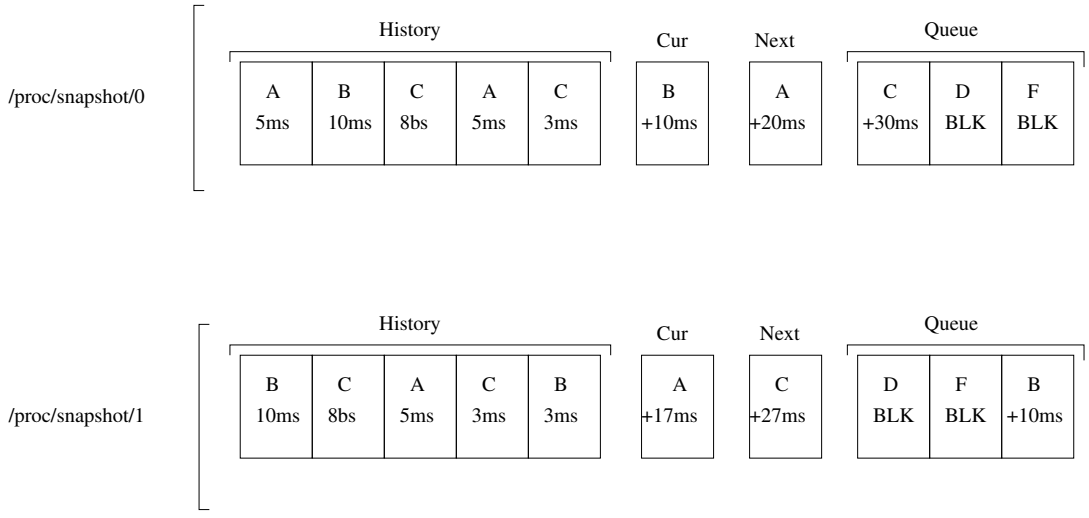

Figure 2: The result of calling `snapshot({_CBS, _CBS}, {_BEFORE, _AFTER}, 2)` on a broken

In order to get the data out of the kernel, you will be implementing a `/proc` interface. You should create the directory `/proc/snapshot` and populate it with `SNAP_MAX_TRIGGERS` files named from 0 to `SNAP_MAX_TRIGGERS` − 1. Each one of these files will represent a snapshot buffer that the user has access to. Reads from these virtual files will return formatted data to the user so they can parse it.

Note that you must implement the functions in `realtime/cbs_proc.h`, which are what we'll be using to implement our `/proc` interface to get data out. Additionally, you must setup you build such that `realtime/cbs_proc_impl.c` gets linked into your kernel image (again, that's how we'll be implementing the `/proc` interface). There are examples of how to cleanly do this in lab 0.

1.3 Deterministicly Testing a Real-Time Process

The end goal of the snapshotting interface is to allow you to test CBS from userspace. There are two important properties of the CBS algorithm that you will need to test: that real-time tasks are being scheduled EDF, and that CBS tasks end up with the correct compute bandwidth in aggregate. The snapshotting interface we've specified allows you to test both of those constraints. It's probably best to take a look at exactly how data is going to move into and out of the VM. Figure 3 shows how the test system is laid out and the interface between each module

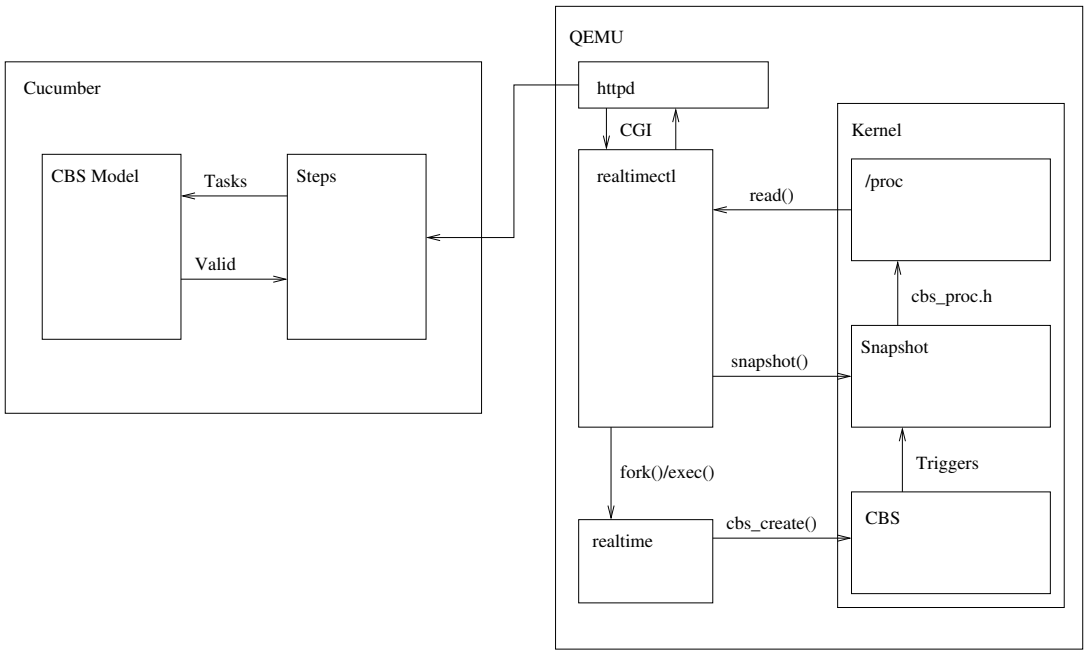

Figure 3: Moving data from CBS to the test harness

You should already have the HTTP server working, and we have provided you with `realtime` and `realtimectl`. You'll want to begin by writing some tests that describe the behavior of the system. An example of a behavior of CBS is shown in Figure 4.

Most of these lines are fairly straight-forward: process creation and snapshoting translate directly into some calls to `realtimectl`. The statement "A should always be scheduled before B", however, is interesting. Effectively this means that your test harness must have a model of CBS that understands when processes should be executed. In this specific case it's simple because all the threads have the same execution period, but in the arbitrary case it'll be significantly more complicated.

Effectively you need a model of CBS that, given a snapshot, is capable of determining if the correct

```
When the following processes are started
   +--------------------------+
   | ID | Type | CPU | Period |
   +--------------------------+
   | A  | RT   | 10  | 1000   |
   | B  | BW   | 90  | 1000   |
   | C  | BW   | 200 | 1000   |
   +--------------------------+
And the system executes for 10 seconds
When any snapshot is taken
Then C should receive 20% of the CPU time
And B should receive 9% of the CPU time
And A should always be scheduled before B
And A should always be scheduled before C
```

Figure 4: A sample behavior of CBS

process is being executed. This will involve looking at the entire list of queued processes and determining that your kernel implementation of CBS has picked the correct process to schedule.

2 Distributed Code

We've distributed some code along with this assignment that's designed to get you started. All this code should be available inside the class's public git repo, in the master branch.

The vast majority of the code provided to you lives inside the `realtime` folder in your git repo.

2.1 `realtime/cbs.[ch]`

These files contain the CBS API that you will have to implement for this assignment along with a userspace implementation of those APIs. Note that the userspace implementation doesn't actually do CBS, it's just the minimal amount of code required in order to actually make code that uses the CBS APIs execute.

The userspace CBS implementation provides no guarantees about execution time at all. You can easily see this: start up a few CPU intensive processes and your real-time tasks will start missing their deadlines. This would be a good comparison for you to make against your proper CBS application: how much background CPU throughput can you get while still hitting your deadlines?

2.2 `realtime/rt,ctl.c`

This contains a pair of applications that are designed to allow you to verify that your real-time deadlines are met. They work as a pair: `realtimectl` is a CGI application that forks to execute `realtime`. Additionally, `realtimectl` is able to use the `snapshot()` interface to get data out of the running kernel. This allows you to leverage your HTTP server to easily test your scheduler, and as such should probably be the first application you take a look at.

2.3 `webvideo`

`webvideo` contains a program that is designed to implement some of the same computational patterns as a video codec. This application already uses our CBS API, so it should require no modification to

run. That said, it will probably be useful to extend `webvideo` to allow you to cover more interesting edge cases.

3 Schedule

Like usual, this assignment will be split into three checkpoints. In order to give you a break during spring break, the first checkpoint has been shortened to be manageable in less than a week.

3.1 Checkpoint 1

The first checkpoint will primarily consist of a design document, along with a little bit of code to demonstrate that you're capable of booting a Linux kernel with a CBS scheduler attached.

Your design document requires the following items:

- (2 points) Summary of the CBS paper linked in this document
- (2 points) Description of the in-kernel data structures required to implement CBS.
- (10 points) A description of how each method in the provided CBS API will be implemented. Be sure to include what will be implemented in userspace and what system calls will be used/modified for each item.
- (2 points) A description of the algorithm you'll be using to implement multi-processor CBS (simpler is better, see checkpoint 3).

Additionally, you must submit code for the following items:

- (5 points) An empty `/proc` entry that will eventually correspond to your snapshot interface.
- (5 points) An implementation of `sched_setscheduler()` that passess `SIGXCPU` to each thread that's been set as a CBS thread as soon as it is scheduled.
- (10 points) A test harness and test cases that allow you to test CBS-scheduled processes, running (and failing some tests) against the provided userspace CBS implementation.

Note that while this checkpoint doesn't require a huge amount of code, you're really going to want to get started on understand the kernel scheduler code as soon as possible. An example of this is the `SIGXCPU` stuff: you're probably not going to have a working CBS scheduler yet, so just hack something into the existing CFS scheduler to make this work.

3.2 Checkpoint 2

The second checpoint consists of the bulk of the assignment: getting a CBS scheduler class implemented in Linux. Unfortunately this is kind of a difficult to split into sub-parts, so it's kind of just listed as a big line item. There are a numbere of smaller behaviors that are required in order for CBS to work, which are also listed.

- (40 points) Single-processor CBS implementation
- (5 points) Send SIGXCPU to over-budget real-time tasks
- (5 points) Don't allow scheduling of a real-time task without execution time
- (5 points) Forward un-unused execution time to CFS for normal tasks

3.3 Checkpoint 3

The third checkpoint consists of a single code requirement: to get multi-processor CBS working. You're allowed to use a very simple algorithm to implement multi-processor CBS: specifically once a thread becomes assigned to a processor it doesn't have to move.

Note that we're also requiring that your snapshot interface works for determinstic testing. This is a pretty major component of the lab, which is why we're essentially dedicating a whole checkpoint to it. Essentially what this bullet means is that you should spend a week testing your single-processor CBS code in addition to making the multi-processor code work (which itself should be fairly simple).

- (10 points) Multi-processor CBS, as described above
- (15 points) Determinstic CBS scheduling