COMP3520 Assignment 1: Scheduling

Due: 11:59 pm on 17/10/2021 AEST

1 COMP3520 Assignment 1

1.1 Task Description

In this assignment you will be implementing two schedulers inside the linux kernel. The first part of this assignment is diving into the kernel code, answering some questions and implementing a simple round robin scheduler. The second part is designing improvements to your scheduler (like changing it to a multi level queue with priority) and critiquing it's design compared to your round robin scheduler and to the existing scheduler in linux.

Each part of the assignment are worth 10 marks each and together the twenty marks represent 25% of your final mark in this course.

1.2 Setup

First, fork the repository for the assignment. If you are unsure how to do that, follow these instructions. Then clone the forked repository:

```
git clone https://github.sydney.edu.au/<your_unikey>/A1
```

There are two branches in the repository, `main` and `linux-cfs-sched` You'll be working in the `main` branch, but for now checkout `linux-cfs-sched`.

1.2.1 Setup fedora docker container

The software in the docker image provided previously is too out of date to build the linux kernel so we'll have to build a new one. Use the provided `Dockerfile` to create a a fedora 34 container for use in assignment. Run the following commands to set it up:

```
docker build -t comp3520a1 A1/
```

You can launch the container using:

```
docker run -it --mount type=bind,source=$PWD/A1,\
   target=/comp3520-a1 comp3520a1:latest zsh
```

On linux you may need to use `sudo` and add the `--privileged` in order for `gdb` to run correctly. If you are in china, follow these instructions here to speed up the downloads.

1.2.2 Booting linux in qemu

Now that we've built the fedora container, we can now compile busybox and the linux kernel. Before the provided `Makefile` will work, we first need to configure the builds for both of them.

Busybox is a program that in one executable contains most of the core utils needed for a system. To configure busybox, run:

```
cd busybox-1.33.1
make defconfig
cd ..
```

To configure the linux kernel on an `x86_64` machine, run the following commands:

```
cd linux-5.10.62/
make tinyconfig
make menuconfig
```

If you are using an M1 mac use:

```
cd linux-5.10.62/
make defconfig
make menuconfig
```

The first `make` command generates the minimal configuration. The second one opens up an `ncurses` based configuration menu in which you can enable/disable more kernel options.

Use this menu to add the configuration options as specified in this article. The order of these instructions is important. Once you've done that enable the following settings in the menu:

- In "General Setup" section, enable:
  - Initial RAM filesystem and RAM disk (initramfs/initrd) support
  - Profiling Support
- In "File Systems/Pseudo filesystems" enable:
  - /proc file system support
  - sysfs file system support
- In the "Executable file formats" section enable:
  - Kernel support for scripts starting with #!
- In the "Device Drivers/Generic Driver Options" section enable:
  - Maintain a devtmpfs filesystem to mount at /dev

- In the "Kernel Hacking" section:
  - In the "Compile Time checks and compiler options" enable
    - Compile kernel with debug info
    - Provide GDB scripts for kernel debugging
  - In the "Scheduler Debugging" section, enable
    - Collect scheduler debugging info
    - Collect scheduler statistics
  - In the "Debug kernel data structures" enable
    - Debug linked list manipulation
  - In the "Tracers" enable
    - Kernel Function Tracer
    - Trace Syscalls
    - Scheduling Latency Tracer
    - Histogram triggers

The config you have created will be stored in `.config`
Finally, run

```
make init
make
```

This will start qemu and you should see kernel booting messages and eventually have a shell that you can use. As a note, you can use the `tinyqemudebug.config` provided instead of doing the above, but I'd recommend going through the process to get comfortable with using `make menuconfig`.

This build may fail if you are on a mac. This is due to the fact that apple filesystem is case insensitive, but the rules to make targets for the kernel are case sensitive. You can get around this by renaming the targets in the `Makefile` to be explicitly lowercase or by using a case sensitive file system (Source)

Your first build will take a while but once it has been built, `make` will only rebuild the objects it has to.

1.2.3 Setup fedora disk image

The busybox/linux run configuration is functional for basic testing, but you may want to have a more fully featured disk image.

You can make a disk image by running the following command inside the docker container:

```
cd /A1
LIBGUESTFS_BACKEND=direct virt-builder fedora-34 --format qcow2\
  --output fedora34.qcow2 --root-password "password:helloworld"
```

This will result in a file called `fedora34.qcow2` to be created. This is essentially a virtual machine image containing it's own kernel and (virtual) disk space. Boot the VM using `make qemu-fed`, login using the username, "root", and the password "helloworld". Finally install the `bcc~/~bpftrace` tools using

```
sudo dnf install bcc bpftrace perf vim wget curl
echo "export PATH=$PATH:/usr/share/bcc/tools/" > .bashrc
```

Once the install finish, you can shut down the VM using `CTRL-a` then hitting `x`.

This image run be it's own inbuilt kernel using `make qemu-fed` or it be can run with *your* kernel with `make qemu-fed-custom-kern`. The tinyconfig doesn't support enough features to boot this image, so you'll have to use the `fedora34.config`

Use this fedora image to profile/test your scheduler using the BCC/BPFTrace tools.

1.3 Part 0: Background Reading

Here are some resources that I recommend you read before diving into the assignment.

- Unreliable Guide To Hacking The Linux Kernel (Guide from the kernel documentation covering common routines and general requirements.)
- A complete guide to Linux process scheduling (Explains all the important data structures in the kernel. Use this with gdb to explore the kernel code). Important sections are:
  - Process (2.1): Overview of `task_struct` and `thread_info` structs
  - Linked List (2.3): Covers the API for linked lists in the kernel
  - Structure (3.1): Covers key data structures used in the scheduling processes, how priorities work and scheduling classes
  - Invoking the scheduler (3.2): Discuss how the scheduler functions are called
  - Skim the rest as required. Understanding CFS may help you with implementing your own priority system for part 2.
- `man 7 sched` Overview of Linux scheduling (The API is not relevant, have a read through the scheduling policies)

- BPF Performance Tools (Available online through USYD library) (Guide to bpftrace, helpful for trace/getting performance stats)
  - Skim Chapter 1 to 5
  - Chapter 6: CPUs
- The Battle of the Schedulers: FreeBSD ULE vs. Linux CFS (Recommended reading for understanding how to test/profile your scheduler)
- Unreliable Guide To Locking (Worth skimming. Crash course on locking in the kernel. Useful if you wish to attempt SMP)

1.4 Part 1: Round Robin [10 marks]

1.4.1 Questions [5 marks]

1. First come first serve [0.5 mark] Is a first come first serve (FCFS) scheduler a good fit for a multi-user operating system? Why? Why not?
2. Initramfs [1 mark] Have a look at the `Makefile` make sure you understand what it does to create the `busybox_initramfs.cpio.gz` file.

   What is purpose of this file? How does it compare to an `initramfs.cpio.gz` taken from a fully featured linux distribution? What role does it play in the boot process? (You can find some documentation in the reference reading section)
3. Scheduler functions and data structures [3.5 marks]

   Make sure you've read "A complete guide to Linux process scheduling" as mentioned above.

   (a) Scheduler Classes and Scheduling Policies [0.5 mark]

   What is the point of scheduler classes like `fair_sched_class` and `rt_sched_class` in the linux kernel? How are they connected to scheduling policies?

   How might you scheduler classes to add or implement your own scheduler in the kernel?

   (b) Follow the Syscall [0.5 mark]

   Which functions in the kernel are responsible for dealing with the system calls fork/exec/exit? How do they tie into the scheduler as a whole? What functions are between the function that handles the system call to a process/task entering/exiting the scheduler run queue?

(c) Key Data Structures [1.5 mark]

What are the important structs used involved in the scheduling process? What information is stored in them? How is that information used in the scheduling process? Make reference to structs used by the CFS scheduler.

(d) CFS Prioritisation [1 mark]

Explain how CFS tracks how long as task has run for and how that information is used in scheduling decisions.

Make reference to fields of the relevant structs.

1.4.2 Code [5 marks]

Taking what you've learnt answering the questions above, implement the `comp3520` scheduler class. You'll mainly be working inside the `comp3520.c` file in the `kernel/sched` directory. You may also want to make some changes to `comp3520_rq` struct in `kernel/sched/sched.h`, or `comp3520_se` in `include/kernel/sched.h`.

You should implement a round robin scheduler, and then write a brief document explain the behaviour of each function in the `comp3520_sched_class` as you have defined it and explain the changes, if any, that you have made to the `comp3520_rq`, `comp3520_se` structs, specifically what data structures you have decided to use.

1.5 Part 2: Building on RR [10 marks]

1.5.1 Code [5 marks]

In this part of the assignment, your job is to improve the round robin scheduler to something better. You are free to design this as you wish, though if you are unsure, you can implement multi level feedback queue scheduling as described in the course textbook (see: reference reading).

1.5.2 Questions [5 marks]

1. Design Decisions

   - Explain the design of your scheduler. Make reference to the structs, their fields and the functions used by the comp3520 scheduler class (Hint: look at the structs `comp352_sched_class`, `sched_comp3520_entity`)
   - How does this scheduler compare to your round robin scheduler?
     - How have you improved on round robin?
     - How do you know it's better? How did you test it? Be sure to explain the rationale behind your test suite.

2. Critique

- What are the limitations of your scheduler? How could it be improved?
- How does your scheduler compare to CFS?
  - Can CFS do anything that your scheduler can't?
  - How did you test the difference in performance characteristics? Be sure to explain the rationale behind your test suite.

1.6 Part 3: Implement Symmetric Multiprocessing (Optional) [5 bonus marks]

This for the students looking for a challenge. If you can implement an SMP Scheduler that works correctly and doesn't deadlock you are eligible for 5 bonus marks.

1.7 Hints and Recommendations

1.7.1 Become friends with GDB

USE GDB! It'll allow you to exploring the code, see what what calls a function that you are interested in and lets look inside structs. It's amazingly powerful. Think of it as repl for C.

To use it with the kernel, run `qemu` with the additional flags "-s -S". This will run a gdb server at port 1234 and will wait till gdb connects before it starts emulation. In the `Makefile`, the recipe `qemu-busybox-debug` does this for you.

Then go the linux source directory and run `gdb vmlinux`. This will run gdb on the kernel. You can then connect to the `gdbserver` running in `qemu`. Consult tutorial 1 for a cheat sheet on how to use `gdb`.

1.7.2 Follow the Syscall

Another way to dig into take a system call that you are familiar with in userspace and trying to find where the code to handle it is in the kernel. Run `grep -R "SYSCALL_DEFINE"` inside the `kernel` folder to see where some major system calls are defined. Then you can set break points there with gdb.

1.7.3 Run QEMU outside the docker if you can

Docker runs containers inside virtual machines on platforms other than linux. Running qemu inside this, may result in a very slow running `qemu` emulation. I recommend that using the docker container to build your programs and then running them inside `qemu` directly in your host os.

You can hardware accelerate qemu by adding the following flags to the `qemu` command:

- On linux: Add the `-accel kvm` flag or use `--enable-kvm`
- On windows: Add the `-accel hax` flag or use `--enable-hax`
- On macos: Add the `-accel hvf` flag

1.7.4 Use cross referencing tools to find definitions/explore

At this link, you can find the bootlin elixir cross referencer. It display source code in a way such that C symbols or preprocessor directives are clickable, and when clicked you are taken to a list containing where it is defined and where it is used. This will be useful when you are using things some of the constructs heavily built on macros such as lists or red black trees.

You can also use `grep -R` to find symbols in the code base.

1.7.5 BPFTrace

Use the BPFTrace/BCC tooling to help you test and profile your scheduler. A very useful program for this purpose `runqlat`.

1.7.6 Consult the other Schedulers

Have a skim of some of the functions in the already existing scheduling classes. It'll give you an idea of what sort of functions that are available or would be simple to set up. For example, there is a simple function that can be acquire a `task_struct` given the `sched_entity`

1.8 Submission

In order to submit the assignment, first create a repository called `COMP3520-A1` under your own account. Then clone run the following commands

```
git clone https://github.sydney.edu.au/COMP3520/A1.git
git remote add origin https://github.sydney.edu.au/<your unikey>/COMP3520-A1.git
git push -u orgin main
git remote add tutor-repo https://github.sydney.edu.au/COMP3520/A1.git
```

The first command clones the tutor repository, the second and third command pushed the cloned repository to the remote repository under your own account. The fourth adds the tutor repository as an additional remote repository, so that you can pull any changes made to the repository.

Once that's been done, add the unikeys, ttho6664 and adha4640, as collaborators to your repository.

Your repository should have 3 branches: `linux-cfs-sched`, `round-robin`, `main`. The latest commit to each branch before the due date, will be marked.

The `main` branch should contain your code and answers to the questions in part 2. The `round-robin` branch should contain your code and answers to the questions in part 1. The `linux-cfs-sched` will be ignored in marking purposes.

Your code should be well commented and your answers to the questions presented along with your critique and analysis should be checked in a pdf file. Ensure that you include any code that you used for testing/profiling purposes in the repository.

You should also submit the PDF to canvas to Turnitin.

1.9 Marking Criteria

The assignment is broken into two parts the Round Robin and it's extension. Each part is worth 10 marks. Each part has both a code section and a written section, each worth 5 marks each.

If your code does not compile, is purposely obfuscated or you will recieve 0 for the code component for the assignment part.

1.10 Academic Honesty

It goes without saying that you are expected to abide by the University of Sydney's academic honesty policy. You are encouraged to discuss the questions, designs, tools and testing methodologies with other students, however everything you submit ***must*** be all your own work.

1.11 Conclusion

Good luck; Have fun!

1.12 Reference Reading

1.12.1 Operating Systems: Three Easy Pieces (Course Textbook)

Recommended for understanding scheduler design

- Scheduling: Introduction
- Scheduling: The Multi-Level Feedback Queue
- Scheduling: Proportional Share
- Multiprocessor Scheduling (Advanced)

1.12.2 Linux Kernel Documentation

- CFS Scheduler
- Linux Tracing Technologies
- Debugging kernel and modules via gdb

- Linux Serial Console
- Red-black Trees (rbtree) in Linux

1.12.3 LWN Articles

- Ftrace: The hidden light switch
- Debugging the kernel using Ftrace - part 1
- Debugging the kernel using Ftrace - part 2
- An introduction to the BPF Compiler Collection
- A thorough introduction to eBPF
- Dumping kernel data structures with BPF

1.12.4 Misc

- Linux tinyconfig and Qemu
- Busybox-based Linux distro from scratch
- Early Userspace in Arch Linux
- Arch boot process
- How initramfs works (Debian)
- Initramfs (Ubuntu)
- Data Structures in the Linux Kernel: Doubly Linked List
- BPF Compiler Collection (BCC)

1.12.5 Papers on scheduling

- The Linux Scheduler: a Decade of Wasted Cores
- ULE: A Modern Scheduler For FreeBSD
- Efficient and Scalable Multiprocessor Fair Scheduling Using Distributed Weighted Round-Robin (Original algorithm for the linux kernel)
- The Battle of the Schedulers: FreeBSD ULE vs. Linux CFS (Recommended reading for understanding how to test/profile your scheduler)
- An Overview of Scheduling in the FreeBSD Kernel

1.13 References

- Linux tinyconfig and Qemu
- Busybox-based Linux distro from scratch
- A complete guide to Linux process scheduling