Homework 4

W4118 Fall 2023

UPDATED: Tuesday 10/31/2023 at 7:00pm EST
DUE: Friday 11/10/2023 at 11:59pm EST

All homework submissions are to be made via **Git**. You must submit a detailed list of references as part your homework submission indicating clearly what sources you referenced for each homework problem. You do not need to cite the course textbooks and instructional staff. All other sources must be cited. Please edit and include this **file** in the top-level directory of your homework submission in the `test` branch of your team repo. Homeworks submitted without this file will not be graded. **Be aware that commits pushed after the deadline will not be considered.** Refer to the homework policy section on the **class web site** for further details.

Programming Problems:

Group programming problems are to be done in your assigned **groups**. The Git repository for your group has been setup already on Github. You don't need the Github Classroom link for the group assignment and do not need to initialize the repository. It can be cloned using:

```
git clone git@github.com:W4118/f23-hmwk4-teamN.git
```

(Replace `teamN` with the name of your team, e.g. `team0`). This repository will be accessible to all members of your team, and all team members are expected to commit (local) and push (update the server) changes / contributions to the repository equally. You should become familiar with team-based shared repository Git commands such as **git-pull**, **git-merge**, and **git-fetch**.

All team members should make at least *five* commits to the team's Git repository. The point is to make incremental changes and use an iterative development cycle. Follow the **Linux kernel coding style** and check your commits with `checkpatch.pl`. Errors or warnings in your submission will cause a deduction of points.

For students on Arm Mac computers (e.g. with M1 or M2 CPU): if you want your submission to be built/tested for Arm, you must create and submit a file called `.armpls` in the top-level directory of your repo; feel free to use the following one-liner:

```
cd "$(git rev-parse --show-toplevel)" && touch .armpls && git add .armpls
&& git commit -m "Arm pls"
```

You should do this first so that this file is present in any code you submit for grading.

For all programming problems you should submit your source code as well as a single README file documenting your files and code for each part. Please do NOT submit kernel images. The README should explain any way in which your solution differs from what was assigned, and any assumptions you made. You are welcome to include a test run in your README showing how your system call works. **It should also state explicitly how each group member contributed to the submission and how many hours each member spent on**

the homework. The README should be placed in the top level directory of the `main` branch of your team repo (on the same level as the linux/ and user/ directories).

Scenario

Having completed the first half of your operating systems training, you and your team are now well-versed OS developers, and have been hired by W4118 Inc. to work on their next-generation operating systems. In this next-gen OS, W4118 Inc. has tasked your team to improve the Linux kernel's scheduling capabilities and optimize for workloads typical of W4118 Inc.'s customers.

Data privacy is important in modern computing, so instead of real customer programs, W4118 Inc. has provided you with a way to simulate the typical workloads submitted by end users using a Fibonacci calculator. The following links provide two task set workloads, each consisting of a list of jobs, with each line representing the N-th Fibonacci number to compute:

1. **TASK SET 1**
2. **TASK SET 2**

Note: The Fibonacci calculator has been provided as `user/fibonacci.c`. It uses an inefficient algorithm by design to produce differing job lengths. You can modify the file, but **do not modify the `fib` function**.

Given these two task sets, W4118 Inc. is interested in a scheduler (let's call it "Oven") that can optimize for the **completion time** of the tasks. Specifically, the scheduler should provide the minimum average completion time for each task set across all tasks in the respective task set.

Let's begin!

1. **Measure scheduler performance with eBPF**

   To see how well a scheduler performs, it is necessary to first have a way to measure performance, specifically completion time of tasks. W4118 Inc. is looking for a way to trace scheduling events and determine how well a scheduler functions by measuring tasks' run queue latency and total duration from start to finish.

   Extended Berkeley Packet Filter (**eBPF**) is a powerful feature of the Linux kernel that allows programs to inject code into the kernel at runtime to gather metrics and observe kernel state. You will use eBPF to trace scheduler events. Tracepoints for these scheduler events are already available in the scheduler (for example, you can search in `core.c` for `trace_sched_`), so your job is to write code that will be injected into the kernel to use them. You should not need to modify any kernel source code files for this first part of the assignment.

   `bpftrace` is a Linux tool that allows using eBPF with a high level, script-like language. Install it on your VM with:

   ```
   sudo apt install bpftrace
   ```

Instead of searching through source code, you can run `bpftrace -l` to see what available tracepoints there are in the kernel.

Write a `bpftrace` script named `trace.bt` to trace how much time a task spends on the run queue, and how much time has elapsed until the task completed. Your profiler should run until stopped with Ctrl-C. While tracing, your script should, at tasks' exits, print out a table of the task name, task PID, milliseconds spent on the run queue, and milliseconds until task completion. Note that task PID here is the actual `pid` field in a `task_struct`, not what is returned by `getpid`. The output should be comma-delimited and follows the format:

```
COMM,PID,RUNQ_MS,TOTAL_MS
myprogram1,5000,2,452
myprogram2,5001,0,15
...
```

Ensure that the output of your eBPF script is synchronous. That is, it should print each line immediately after a task completes, and not when Ctrl-C is sent to the eBPF script. You should also only print traces from processes that have started after your script begins running. You should trace all such process and perform no additional filtering.

Test your script by running `sudo bpftrace trace.bt` in one terminal. In a separate terminal, run a few commands and observe the trace metrics in your first terminal. Experiment with different task sizes. Submit your eBPF script as `user/trace.bt`.

Now that you have an eBPF measurement tool, use it to measure the completion times for the the two task set workloads when running on the default CFS scheduler in Linux. What is the resulting average completion time for each workload? Write the average completion times of both workloads in your README. You may find this **shell script** helpful for running your tasks. You should perform your measurements for two different VM configurations, a single CPU VM and a four CPU VM. We are using the term CPU here to mean a core, so if your hardware has a single CPU but four cores, that should be sufficient for running a four CPU VM. If your hardware does not support four cores, you may instead run with a two CPU VM. Specify in your README the number of CPUs used with your VM for your measurements.

Hint: You may find it useful to reference the `bpftrace` **GitHub project**, which contains a manual and a list of sample scripts. A useful example script to start with is `runqlat.bt`.

Note: Since your profiler will run on the same machine as the test workloads, you will need to ensure that the profiler gets sufficient CPU time to record data. What could you do to the profiler's scheduling class and priority to ensure it can run over the test workloads? You may find `chrt` helpful.

2. **Create your scheduler**

W4118 Inc. has tasked you with creating a new scheduling policy that provides better average completion time than the default Linux scheduling policy. You should begin by implement a new scheduler policy. Call this policy `OVEN`. For this stage, the scheduler should work as follows:

i. Only tasks whose policy is set to `SCHED_OVEN` should be considered for selection by your new scheduler.

ii. Every task will have an assigned weight. Weights should be configured when a process calls the `sched_setscheduler` system call, and passed as the `sched_priority` field of `struct sched_param`.

iii. By default, tasks have a weight set to a MAX_WEIGHT of 10000. Tasks with a weight equal to the MAX_WEIGHT of 10000 should use an unweighted round-robin scheduling algorithm. Each such task should get a time quantum of 1 tick.

iv. Tasks with a weight less than 10000 should be prioritzed over weights equal to MAX_WEIGHT, and use your choice of scheduling algorithm and time quantum to optimize the overall average completion time metric for running the Fibonacci workload. Essentially, your scheduling class defaults to round-robin but should have a special mode that is optimized for running the Fibonnaci workload, where the optimized scheduling algorithm is up to you to decide.

v. Tasks using the `SCHED_OVEN` policy should take priority over tasks using the `SCHED_NORMAL` policy, but *not* over tasks using the `SCHED_RR` or `SCHED_FIFO` policies.

vi. The new scheduling policy should operate alongside the existing Linux schedulers. The value of `SCHED_OVEN` should be 7.

Some notes that may be helpful in your implementation:

- The Linux scheduler implements individual scheduling classes corresponding to different scheduling policies. For this assignment, you need to create a new scheduling class, `oven_sched_class`, for the `OVEN` policy, and implement the necessary functions in `kernel/sched/oven.c`.
- A good starting point is the simple scheduling class for idle tasks, listed in `kernel/sched/idle.c`. This is the simplest of the scheduling classes, so is a good place to start. In particular, the functions implemented by this scheduling class are a good indication of the minimum set of functions you need to implement to have an operational scheduling class. Additional examples of scheduling classes can be found in `kernel/sched/rt.c` and `kernel/sched/fair.c`. You may find the former particularly useful because it has its own version of a round-robin scheduler, SCHED_RR, though you will likely find it too complex to use directly for your own scheduling class.
- Other interesting files that will help you understand how the Linux scheduler works are `kernel/sched/core.c`, `kernel/sched/sched.h`, `include/linux/sched.h`, and `include/asm-generic/vmlinux.lds.h`. While there is a fair amount of code in these files, a key goal of this assignment is for you to understand how to abstract the scheduler code so that you learn in detail the parts of the scheduler that are crucial for this assignment and ignore the parts that are not.
- Although Linux in theory allows you to define new scheduling classes, Linux makes assumptions in various places that the only scheduling classes are the ones that have already been defined. In particular, there is code in

`kernel/sched/core.c` that makes assumptions about the prioritization of scheduling classes, including `sched_init` and `__pick_next_task`. Review those functions carefully. Furthermore, other functions may also need to be changed to allow you to assign tasks to your scheduling class, such as those called by `sched_setscheduler`, and allow your scheduling class to be used to pick the next task to run. You should pay careful attention to how scheduling classes initialize their class-specific run queues and scheduling entities to insert them onto the run queues in `kernel/sched/core.c`; incorrect initialization can easily cause your system to hang. To identify which parts need to be changed, look for references to existing Linux schedulers, particularly in the files mentioned above. Pay special attention to areas that explicitly reference the current default CFS scheduler (`fair_sched_class`).

- While developing and debugging your initial scheduling class implementation, it will be helpful to initially have `SCHED_NORMAL` take priority over your `SCHED_OVEN` policy so that bugs in your scheduling class are less likely to prevent other tasks from running. Once you have your scheduling class working, you can then switch the priority of these policies as required by the homework.
- Implementing a scheduler and getting everything right is not easy. You should make your implementation as simple as possible. Avoid using complex data structures that may provide better theoretical runtime complexity but are harder to implement and debug. Lists are in general fine to use.
- Test your scheduler on a few runs of `fibonacci` and ensure it works before moving on. Set the weight to something other than the default and verify its behavior. After verifying that Fibonacci works, try running the **program from homework 3 that tests state changes**. The latter will exercise your scheduling code more throughly as it will involve scheduling processes that have a greater variety of state changes.
- Once your scheduler works for both `fibonacci` and the state program from homework 3, set it to be the default scheduler of your system. You will need to change various places in the kernel code to allow your kernel to boot with your scheduling class as the default. You should consider how a task is assigned to a scheduling class by default.
- You may want your scheduler to have a minimum valid weight well above the priority range normally used for `sched_priority` such that any weight assignment less than the minimum valid weight is instead forced to be the MAX_WEIGHT. The reason for this is that there are system programs that use `sched_setscheduler` with lower values for `sched_priority`. If they are scheduled using your scheduling class, you want to detect their invalid weight assignments and schedule them using your round-robin algorithm.
- For a more responsive system, you may want to set the scheduler of kernel threads to be `SCHED_OVEN` as well (otherwise, `SCHED_OVEN` tasks can starve the `SCHED_NORMAL` tasks to a degree). To do this, you can modify `kernel/kthread.c` and replace `SCHED_NORMAL` with `SCHED_OVEN`. It is strongly suggested that you do this to ensure that your VM is responsive enough for the test cases, but you should not do this until you are certain your scheduler works properly.
- Check out the **debugging tips** provided below.

3. **Optimizing and measuring your new scheduler**

   Recall your average completion time measurements from part 1, which used the CFS scheduler. Can you do better?

   You should consider how you might modify the `fibonacci` program to set its OVEN weight. How did you determine when jobs of different weights will be scheduled in relation to each other? What weights and scheduling algorithm will optimize for average completion time? Make any changes to `fibonacci` and your scheduler, then submit eBPF traces of the two task sets running on your scheduler. You should use the same two VM CPU configurations you used earlier to measure performance using CFS. Submit your eBPF traces for the two task sets and CPU configurations as `user/taskset1_average.txt`, `user/taskset1_average_smp.txt`, `user/taskset2_average.txt`, and `user/taskset2_average_smp.txt` Write the average completion times for each workload in your README and compare against CFS.

   Hint: Configuring the scheduling class and priority within the Fibonacci program will be too late (why is this the case?). You may want to write a small program to set the scheduler and weight before calling exec on Fibonacci.

   Note: When launching jobs from the task list, start tasks in the order they are listed, but do not wait for tasks to finish before launching the next one. Also, for all trace submissions in this section, filter out any process that is not `fibonacci`.

4. **Add load-balancing features**

   So far, your scheduler will run a task only on the CPU that it was originally assigned. Let's change that now! For this part you will be implementing idle balancing, which is when a CPU will attempt to steal a task from another CPU when it doesn't have any tasks to run (i.e. when its runqueue is empty).

   Load balancing is a key feature of the default Linux scheduler (Completely Fair Scheduler). While CFS follows a rather complicated policy to balance tasks and works to move more than one task in a single go, your scheduler will have a simple policy in this regard.

   Idle balancing works as follows:

   - Idle balance is when an idle CPU (i.e. a CPU with an empty runqueue) pulls one task from another CPU.
   - Take a look at the CFS implementation to figure out where this is taking place and how it is called from `kernel/sched/core.c`.
   - The CPU that you pull a task from should have at least two tasks on its runqueue. Which task you pull off is up to your scheduling algorithm.
   - You should again ensure that the task you are stealing is eligible to be moved and allowed to run on the idle CPU.

   Once you add idle load balancing, repeat the performance tests you conducted in the previous part and make notes on any observations. Again, be

sure to include actual data. Submit the trace in `user/taskset1_smp_balance.txt` and `user/taskset2_smp_balance.txt` and note any differences to your previous scheduler in your README. You only need to submit results in this case for the same multi-CPU VM configuration as you used previously, but do not need to submit single-CPU VM results for this case. Write the updated average completion times for each workload in your README and compare against CFS.

5. **Tail completion time**

   Thus far you have focused on optimizing the average completion time across all jobs. Another important metric to consider is tail completion time, which is the maximum completion time across 99% of all jobs. Tail completion time focuses on ensuring that the completion time of most jobs is no worse than some amount. Using tail completion time as the performance metric of interest, compare your optimized scheduling class versus the default Linux scheduler for the W4118 Inc. workloads. Which one does better? If the default Linux scheduler has better tail completion time, can you change how you use your OVEN scheduler (without modifying the OVEN scheduler code) to provide tail completion time comparable to the default Linux scheduler?

   Make any changes to `fibonacci` , then submit eBPF traces of the two task sets running on your scheduler as `user/taskset1_tail.txt` and `user/taskset2_tail.txt`. You only need to submit results in this case for the same multi-CPU VM configuration as you used previously, but do not need to submit single-CPU VM results for this case. Write the tail completion times for each workload in your README and compare against CFS.

6. **Analysis and investigation of kernel source code**

   Write answers to the following questions in the `user/part6.txt` text file, following the provided template exactly. Make sure to include any references you use in your references.txt file.

   1. Give the exact URL on elixir.bootlin.com pointing to the file and line number of the function that initializes the idle tasks on CPUs other than the boot CPU for a multi-CPU system. What is the PID of the task that calls this function? Note: make sure you use v6.1.11.
   2. Give the exact URL on elixir.bootlin.com pointing to the file and line number at which the TIF_NEED_RESCHED flag is set for the currently running task as a result of its time quantum expiring if the task is scheduled using SCHED_RR. Select the file and line number most related to the SCHED_RR (i.e. do not select a generic helper function that may be used outside of SCHED_RR). Note: make sure you use v6.1.11.
   3. Give the exact URL on elixir.bootlin.com pointing to the files and line numbers at which a timer interrupt occurring for a process running in user mode, whose time quantum has expired, results in schedule being called. Select the line of the call to schedule. This location is different for ARM64 and x86 - provide the answer for both. Note: make sure you use v6.1.11.
   4. What is the default time period for a tick? Write your answer in milliseconds. Hint: You may need to look in the kernel configuration files

to find this answer.

Debugging Tips

The following are some debugging tips you may find useful as you create your new scheduling class.

- Before testing, take a snapshot of your VM if you have not done so already. That way, if your kernel crashes or is unresponsive because of scheduler malfunction, you can restore the state of the VM prior to the problem.
- It is possible that your kernel gets stuck at boot time, preventing you from reading debug messages via `dmesg`. In this scenario, you may find it helpful to redirect your debug messages to a file on your host machine.
  1. Right click on your VM and click "Settings"
  2. Under "Hardware", click "Add" and create a Serial Port.
  3. Under "Device", click on your new Serial Port.
  4. Click on "Use output file", and specify the file on your host machine to which your would like to dump the kernel log.
  5. Turn on your VM, move to your kernel in the GRUB menu, and press `e`.
  6. Move toward the bottom until you find a line that looks like (`linux /boot/vmlinuz-6.1.11-cs4118...`).
  7. At the end of this line, replace `quiet` with `console=ttyS0` (try `console=ttyS1` if this doesn't work).
  8. Hit `F10` to boot your kernel. The kernel log should be written to the file on your host machine you specified earlier.
  9. If neither `ttyS0` nor `ttyS1` work, you may need to remove the virtual printer hardware in your VMware VM settings.
- You may find it helpful to use `ps`, `top`, or `htop` to view information such as the CPU usage and scheduling policies of your tasks. These commands rely on execution time statistics reported by certain scheduler functions in the kernel. As a result, bugs in your scheduling class could cause these commands to report inaccurate information. This cuts two ways. First, it is possible that your scheduler is working properly in terms of selecting the right tasks to execute for the right amount of time, but your calculation of execution time statistics in the kernel is wrong, so these commands appear to report that your tasks are not running at all when in fact they are. Second, it is possible that your scheduler is not working properly such that these tools report that tasks are using your scheduling class when in fact they are not.

  As a result, you should not exclusively rely on these tools to determine if your scheduling class is working properly. For example, when you make your scheduling class the default scheduling class, the fact that user-level tools claim that all tasks are using your scheduling class may not necessarily mean that this is the case. Instead, to ensure that your scheduling class functions are actually being used, you might add a `printk` to a function like your class's `enqueue_task()` and verify that it appears in `dmesg` output. Make sure that you

do not submit your code with such debugging `printk` statements as they can cause issues if invoked too frequently.

Submission Checklist

Include the following in your main branch. Only include source code (ie *.c,*.h) and text files, do **not** include compiled objects.

- `.armpls` file for teams with M1/M2 CPUs
- `README` file that includes a description of your scheduling algorithm and how you do idle load balancing, the number of cores enabled in the VM you used for your measurements, and a discussion of your completion time results compared to CFS.
- `references.txt` file
- Implementation of your scheduler in `linux/kernel/sched/oven.c`
- Changes to `fibonacci.c`
- Implementation of `trace.bt` in `user/trace.bt`
- Output of running your scheduler on the workloads in `user/taskset1_average.txt`, `user/taskset2_average.txt`, `user/taskset1_average_smp.txt`, `user/taskset2_average_smp.txt`, `user/taskset1_smp_balance.txt`, `user/taskset2_smp_balance.txt`, `user/taskset1_tail.txt`, and `user/taskset2_tail.txt`.
- Answers to written questions in `user/part6.txt`