Coursework 1

INFR100792023 (Operating Systems) [2025-26 Spring]

February 1, 2026

1 Task 1: A Shady Job [30 points]

Recently, a rare hardware security vulnerability was exposed in many CPU models manufactured by *Outel Technologies*. The security vulnerability was described in a top secret internal report, from which the following extract was leaked to us confidentially.

> *... don't know which ones exactly,* ***[REDACTED]*** *for sure. We had noticed this way back in testing phase. Since we got the lithography wavelengths to below 1 nanometer, weird quantum entanglement effects started popping up between certain CPU cores, which break inter-core isolation guarantees. So suppose if a core 6 and core 7 are entangled, then all data belonging to a process on core 6 will become visible to the a process on core 7. I mean, this is a* major *vulnerability. We filed these findings and presented them to* ***[REDACTED]****, but I don't think they understood how big the issue was... or maybe they did but decided to keep it anyway, as the cost of doing a recall is always greater than the ...*

Officials from *Outel Technologies Edinburgh* have reached out to us, to help apply an OS security patch to hide this security vulnerability. We ask your help in implementing this patch for Linux, for which you will be compensated with grade points.

1.1 Problem Statement

You are given a *pair of entangled CPUs* (X, Y). Implement cross-CPU mutual exclusion in CFS (Completely Fair Scheduler) for cores X and Y based on the user. A process belonging to a user ID *uid* should not be scheduled on CPU X if a process belonging to another user (other than *uid*) is currently running on Y, and vice-versa for Y in place of X. Limit all your modifications to `kernel/sched/fair.c`.

1.2 Specifications for Implementation

You are expected to implement your solution on Linux v6.12.67 [1]. You must first apply the following two edits to `kernel/sched/fair.c`. First, you must define two `unsigned int` variables named `entangled_cpus_1` and `entangled_cpus_2`. Second, find the definition of the array `sched_fair_sysctls` in the same file, and add the following two entries corresponding the entangled CPU IDs.

```
static unsigned int sysctl_entangled_cpu1 = 0;
static unsigned int sysctl_entangled_cpu2 = 0;
...
static struct ctl_table sched_fair_sysctls[] = {
#ifdef CONFIG_CFS_BANDWIDTH
...
#endif /* CONFIG_NUMA_BALANCING */
{
               .procname = "entangled_cpus_1",
               .data = &sysctl_entangled_cpu1,
               .maxlen = sizeof(unsigned int),
               .mode = 0644,
               .proc_handler = proc_dointvec_minmax,
},
{
               .procname = "entangled_cpus_2",
               .data = &sysctl_entangled_cpu2,
               .maxlen = sizeof(unsigned int),
               .mode = 0644,
               .proc_handler = proc_dointvec_minmax,
},
};
```

The above ensures the creation of a *procfs* interface that can be used to set the IDs of the entangled cores. After building and booting into the new kernel, check the contents of `/proc/sys/kernel`. You will find that two new files named `entangled_cpus_1` and `entangled_cpus_2` have been created. These files are linked to the variables `sysctl_entangled_cpu1` and `sysctl_entangled_cpu2` that you have defined inside `kernel/sched/fair.c`.

You can use this interface as per the following example. To set $(1, 3)$ as the entangled pair, the following steps may be taken.

```
USER@HOST$echo 1 > /proc/sys/kernel/entangled_cpus_1
USER@HOST$echo 3 > /proc/sys/kernel/entangled_cpus_2
```

The values set in this way can be accessed in `fair.c` via the variables defined earlier.

```
int c1 = sysctl_entangled_cpu1;
int c1_new = READ_ONCE(sysctl_entangled_cpu1); // gets the fresh (latest)
     value of sysctl_entangled_cpu1
```

[1] `https://cdn.kernel.org/pub/linux/kernel/v6.x/linux-6.12.67.tar.xz`

If `entangled_cpus_1` and `entangled_cpus_2` are set to the same value, then that means that there is no entanglement. By default, both are set to 0.

NEW CONSTRAINT. You must ensure that no CPU is left continuously idle for 10 seconds of wall-clock time while there exists a runnable task on the system.

2 Task 2: *Completely* Fair Scheduler [30+40 points]

Comrade! You are once again called upon to contribute your programming skills to the glory of the revolution! As you know, our party started a public cloud server, so that any member of the working class may run their programs on the cloud without relying on bourgeoisie capitalist alternatives like AWS. This service was a great success on launch.

However, bourgeoisie tendencies have now crept back in, and we need your help to eradicate them. It has been observed that some users hog all available CPU time. They make use of dirty tricks to cling on to surplus CPU time - for example, if their program need to wait for a 5 seconds, instead of releasing the CPU to the scheduler via `sleep()`, they use an empty while loop with no body, keeping the CPU busy. Your mission, comrade, is to ensure that every user on a system gets the same total CPU time. First, you must identify the users who hog most resources, so that we may blacklist them. Second, you must modify the Operating System so that every user gets equal CPU time, whenever there is a scarcity of available CPU time. May you succeed in your mission!

2.1 Problem Statement

The problem statement has two components, to be graded independently of each other.

2.1.1 SubTask A: User-Space CPU usage Monitor [30 points]

Implement a user-space program that

1. Periodically monitors CPU usage of all running processes.
2. Aggregates CPU usage per user.
3. Produces a ranked list of users based on their total CPU consumption.

You must write your code in C. You must include a `Makefile` [2] that contains instructions to build your code into an executable file called `monitor.exe`. You must ensure that on a system that supports GCC 13.0+, invoking `make` builds your code, and once built, `make clean` removes all the generated files. The

[2] `https://www.gnu.org/software/make/manual/make.html`

program must be invoked with one command-line argument that represents the number of seconds for which the program must run. At the end of this duration, it should print the final output to stdout.

The following is an example of a source code structure that you may use.

```
USER@HOST$ls
Makefile monitor.c utils.c utils.h
USER@HOST$make
USER@HOST$ls
Makefile monitor.c monitor.exe monitor.o utils.c utils.h utils.o
USER@HOST$./monitor.exe 30 # runs for 30 secs and prints output at end
USER@HOST$make clean
USER@HOST$ls
Makefile monitor.c utils.c utils.h
```

Once invoked, `monitor.exe` must do the following *once every second* during its lifetime:

1. Enumerate all active process IDs (PIDs) using the `/proc` filesystem.
2. For each PID, read CPU usage information from `/proc/<pid>/stat` [3].
3. Compute per-process CPU time (both user and kernel), and attribute it to the user of the process. Aggregate CPU usage time per user.

At the end of the prescribed duration, the program should print a summary similar to:

```
Rank  User        CPU Time (milliseconds)
----------------------------------------
1     alice       12543
2     bob          9821
3     charlie      4310
```

The summary should only count running times since the start of `monitor.exe` - so if any process had been running before `monitor.exe` started, you should disregard the CPU time consumed before the start of `monitor.exe`.

We will test your submission on a system running Linux v6.12.67 (unmodified) as the OS kernel.

2.1.2 SubTask B: Equitable Scheduling Across Users [40 points]

Modify the Linux Completely Fair Scheduler (CFS) to enforce *equitable CPU allocation across users*.

The goal is to ensure that each user receives approximately the same fraction of *non-idle CPU time* over real wall-clock time, independent of the number of processes or threads that user runs. Modify the CFS scheduler so that runnable

[3] `https://man7.org/linux/man-pages/man5/proc_pid_stat.5.html`

tasks are *grouped by user*, and CPU time is distributed equitably across users rather than tasks.

Note that

- You only need to consider users with UID ≥ 1000.
- Only runnable (non-idle) CPU time should be considered when evaluating fairness.

Your scheduler should satisfy the following properties:

- **Equity across users:** Over time, each user should receive approximately the same fraction of CPU time, regardless of how many runnable tasks they have. The fraction of CPU time is computed as the ratio between total time the process spent running on CPU and the real wall-clock time.
- **No unnecessary throttling:** When there is no CPU scarcity (i.e., the number of runnable tasks is less than or equal to the number of available CPUs), all runnable tasks should be allowed to run without restriction.

You may use any reasonable strategy to achieve equitable scheduling across users. Possible approaches include (but are not limited to):

- A greedy strategy that prioritizes users with the lowest CPU fraction every scheduling tick.
- Periodic accounting of CPU usage per user, followed by task selection biased toward under-served users.
- Dynamic adjustment of task weights or virtual runtimes based on per-user CPU consumption.

You are not required to achieve perfect equality at all times. Instead, your goal is to demonstrate a policy that converges toward equitable CPU distribution across users under sustained contention.

Evaluation Your implementation will be evaluated based on:

- Effectiveness in balancing CPU time across users under contention.
- Preservation of system responsiveness and avoidance of starvation.

Deliverables You must submit:

- You may modify any source file within `kernel/sched`, though `fair.c` and `core.c` should suffice for most implementations.
- You should implement your work on Linux `v6.12.67` [4]. Although this is the same kernel version as Task 1, you must not include the edits we have requested in 1.2 for this task. You should attempt this task on a fresh instance of Linux v6.12.67 source code.

[4] `https://www.kernel.org/`

3 Submission

3.1 Deliverables

You should submit a .zip archive with your student ID (UUN, starting with S) in its name, in the format `CW1_UUN`. The archive must contain a folder with the same name. The folder must contain one sub-folder each for Task 1, Task 2A, and Task 2B.

1. Task1 contains your modified `fair.c` file for Task 1. This file is the `kernel/sched/fair.c` file from Linux v6.12.67 including the modifications recommended in 1.2 and your solution for Task 1.
2. Task2A contains your user-space source code as described in 2.1.1.
3. Task2B contains your modified files from `kernel/sched` for Task 2B. These files are from Linux v6.12.67. This is the same version as Task 1 kernel, but this should NOT include the edits from 2.1.1.

The following is an example directory structure you may use.

```
USER@HOST$ls CW1_S1234762
Task1 Task2A Task2B
USER@HOST$ls CW1_S1234762/Task1
fair.c
USER@HOST$ls CW1_S1234762/Task2A
Makefile monitor.c
USER@HOST$ls CW1_S1234762/Task2B
fair.c core.c
USER@HOST$zip -r CW1_S1234762.zip CW1_S1234762
```

Submit the .zip archive to the LEARN portal.

3.2 AI Usage Policy

If you used AI to generate any source code you should make sure that such code works, which means it does compile and it executes as expected — i.e., the system runs without any error. It is not the responsibility of AI to make a specific code compile and produce the correct results — it is the student's responsibility. You may add a TXT file inside your submitted folder to make any admissions related to AI usage.

3.3 Grading Policies

Your code will be auto-tested on a wide range of test-cases. Some test-cases for each task will be made public in due course of time. Make sure that your code compiles without errors on DICE. Any submission that does not compile is eligible to get 0 points. Passing all test-cases guarantees full marks, while passing *some* test-cases generally secures *some* marks. In addition to this, the graders may award partial marks on their discretion.