COMP3520 Assignment 1

Student Name: Kang-Min Seo (preferred: Calvin Seo)
SID: 480347192
Unikey: kseo6751
Tutor name: Tyson, Anuj
Assignment Name: COMP3520 Assignment 1

DUE: 19.10.2021

Contents

0. Overview

0.1 The spec of the computer that was used for this assignment

Model Name: Thinkpad T14 Gen 1(amd)
CPU: AMD Ryzen 5 PRO
4650U 6 cores, 12threads 2.1GHz 4.0GHz 3MB L2 / 8MB
AMD Radeon Graphics

OS: Ubuntu 20.04

0.2 Very brief overview of work

Initially, I was trying to implement this assignment using MacOS. However, I realized that I had to put extensive efforts to fix some of the errors. Therefore I have moved to a different OS and started implementing.

The computer was running Ubuntu 20.04 and tasks were finished from the docker container. All of the round-robin and multi-level feedback queue has been implemented successfully. I have tested this by following the instructions on the spec sheet and at the end did make init, make to get into the kernel. As a result, both of my schedulers worked and were successful in running some of the test programs that were provided.

I have used the GDB to ensure that my scheduler functions are being called correctly, and manage the queue (from the multi-level feedback queue) as desired. I did this by following the instructions. I ran "make qemu-busybox-debug" and from another tmux window, I ran "gdb vmlinux" inside the Linux folder. I connected them by 'target remote localhost:1234' and set breakpoints to my functions and observed their behaviors.

For creating the histogram of runq latency, I have not made it outside of the docker container using a custom kernel. Meaning I have downloaded qemu and ran the command and was able to successfully access the kernel. I ran the runqlat 60 1 -P and found the task I was running with the process id.

MESSAGE TO MY MARKER (Please Read this)
First of all, thank you very much for marking my assignment. I am writing this message to apologize for doing a late submission. I am submitting this on the 21st of October (Thur) which is two days after the deadline (10% reduction). My sister has a disease and her symptoms got worst recently and I was very depressed throughout the semester as I couldn't do anything since we are staying apart. I think this has affected my time management in general. I didn't want to submit a special consideration since then it will ask me for proof and I wouldn't want to ask her or my parents for her hospital record since they are also very depressed. I am not writing this message to make you feel sympathy for me, but the reason why I am letting you know this is because I am truly sorry about this late submission which might affect your marking schedule. It is my first time doing a late submission, and I just wanted to let you know there was something beyond my control going on.

1. Part 1 Round-Robin [10 mark]

1.1 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?

First come first serve is not a good fit for a multi-user operating system. Here is why. As from the name first come first serve schedules the task by the incoming order. If such a task is a batch process meaning that it requires a lot of CPU time then, the next task that arrives might have to wait a long time. And in the case of a multi-user operating system that task might be from another user resulting in a very bad "response time". Such behavior is undesirable since it is not fair to the other processes.

1.2 Initramfs [1 mark] Have a look at the Makefile to make sure you understand what it does to create the busybox_initramfs.cpio.gz file. ...

initramfs is an abbreviation of the initial Ram file system. It is the archive of Cpio in the initial file system. This file system is temporary where it contains all the drivers, programs, and binary. It is necessary for the kernel's initialization during the booting process. The busybox initramfs is on the high overview, our custom-built initramfs that helps during the booting process. We are using the init.sh to boot up the program. initramsfs allows the kernel booting process to include userspace, which in the busy boxes case includes some of the testing programs inside test_suits.

1.3 Scheduler functions and data structures [3.5 marks]

1.3.1 Scheduler Classes and Scheduling Policies [0.5 mark]

sched_class is defined in sched.h inside the kernel repository. It includes all the function pointers to the necessary functions for the scheduler. This includes enqueue task, dequeue task, yield task, and many more. It is just like an interface of java where fair_sched_class, rt_sched_class are the instances of this shed_class with their implementation of the function pointers based on their scheduling policies. Specific implementation can be seen inside fair.c and rt.c for fair_sched_class and rt_sched_class respectively. In the sched repository, it also has a file called core.c. And based on the task's priority number it sets the appropriate sched class to the task. Therefore we need to modify the __setscheduler_prio function from core.c which has the functionality of initializing the task's scheduler class. In our case since our round-robin and multi-level feedback, the queue is for the normal tasks (or at least I've decided) we need to replace the scheduling class for the original CFS. Then we need to implement those function pointers declared in sched_class.

1.3.2 Follow the Syscall [0.5 mark]

When the fork system call happens is that it calls the kernel_clone(). This is inside the kernel repository under the fork.c. Inside this function, it calls the copy_process which creates the copy of the registers and all the other appropriate parts. However, this will not yet start the process. Instead, it calls the sched_fork function which is defined in our core.c. It will then call __set_task_cpu which will instruct the CPU to load the tasks' run queue into the CPU's run queue.

For the exec system call, if it enters, it will call do_execve then call the do execveat_common then call the bprm_execve, and finally it reaches the core.c and utilize the function sched_exec. And inside the sched_exec is responsible for balancing the CPU. It came to my attention that it is not

Last but not least we have the exit system call. It is done by calling do_exit() then it will call the do_task_dead which will change the task flag to dead which later when the schedule() function is called it will be dequeued from the run queue of the CPU.

(Note: I have answered this using the elixir bootlin website to navigate through the functions and structures)

1.3.3 Key Data Structures [1.5 mark]

During the implementation process of my round-robin scheduler, five main structs were heavily involved *(It will be underlined)*. This would also apply to the CFS since the structures were utilized similarly. First, we have the **struct rq**. This is also known as per core run queue and it holds the run queue for each scheduling policy. Inside this struct, there is a field for cfs_rq. **struct cfs_rq** is the next very important structure. This will hold the **struct shed_entity** which will be heavily utilized to navigate to the next available task in the queue. However, the **important** fact is since **struct task_struct** is a very large structure (as it contains all the necessary information for the tasks), it would have a huge overhead if it had to iterate through this structure to manage the queue. The Linux scheduler was able to solve this in a counterintuitive way. With our intuition, the linked list would have a pointer to the next node and in each node, it would have its unique field. However, since these fields are huge (task_struct) it would be inefficient to iterate through a large structure. To minimize this, sched_entity only holds the **struct list_head** group_node, which only contains a pointer to the next and prev list_head. Here it doesn't contain any information about shed_entity or task_struct. Since struct sched_entity had its list_head field, it can be found the structure that is containing this list_head, which in this case would be shed_entity. (using the container_of function or list_entry) Then we can also find the task_struct since sched_entity is also a field of task_struct using the same functions. To summarize, to make the memory usage more efficient, instead of embedding task_struct into the sched_entity, it uses struct list_head group_node to navigate and potentially be used to find the sched_entity and task_struct.

1.3.4 CFS Prioritisation [1 mark]

Inside the sched_entity of the CFS, there is a member called vruntime which is virtual runtime. . CFS will let the task with the lowest runtime. It is considering the weight of the process to other tasks that are on the queue by considering how much time the task has already spent on the CPU. If it has a low vruntime it means that such task needs the CPU and has a higher priority comparably to the other process.

1.4 Code [5 mark]

I have not made any changes to the important data structures, that were originally provided to us. This includes comp3520_rq, sched_comp3520_entity. For the round-robin, I have implemented enqueue_task_comp3520, dqeueue_task_comp3520, pick_next_task_comp3520 and

task_tick_comp3520. Comp3520_rq will store the comp3520 entity wherein the entity will include list_head structure which later on can be used to find the appropriate struct task_struct. This is crucial since task_struct contains all the necessary information for the tasks to run. En queue task will handle these and put the new task on the queue. Dequeue will dequeue the task from the queue. Task_tick will be called every predefined interval time wherein my case I have called resched_curr(rq) functions. This will ensure pick_next_task to be called to find the appropriate next task and run them.

These are the benchmark result of running the test.c inside test_suit repository

```
pid = 232
     usecs               : count     distribution
         0 -> 1          : 1752815  |****************************************|
         2 -> 3          : 46900    |*                                       |
         4 -> 7          : 7124     |                                        |
         8 -> 15         : 1773     |                                        |
        16 -> 31         : 697995   |***************                         |
        32 -> 63         : 27599    |                                        |
        64 -> 127        : 6376     |                                        |
       128 -> 255        : 9656     |                                        |
       256 -> 511        : 18302    |                                        |
       512 -> 1023       : 24653    |                                        |
      1024 -> 2047       : 39099    |                                        |
      2048 -> 4095       : 3979     |                                        |
      4096 -> 8191       : 337      |                                        |
      8192 -> 16383      : 83       |                                        |
     16384 -> 32767      : 7        |                                        |
```

Original CFS runqlat

```
pid = 230
     usecs               : count     distribution
         0 -> 1          : 1995694  |****************************************|
         2 -> 3          : 221      |                                        |
         4 -> 7          : 124      |                                        |
         8 -> 15         : 73       |                                        |
        16 -> 31         : 46       |                                        |
        32 -> 63         : 21       |                                        |
        64 -> 127        : 4        |                                        |
```

Round-Robin runqlat

2. Part 2 Multi-Level feedback queue [10 mark]

2.1 Design Decisions

2.1.1 Explain the design of your scheduler

The design of the scheduler followed the five rules that were mentioned in the book which were if the Priority of task A is bigger than Priority B task A should run. If they have the same priority then they should run in a round-robin fashion. When the new task C comes in it will have the highest priority queue. And when each job stays a certain amount of time in the queue its priority should be reduced and moved to a lower priority queue. And finally, all tasks get promoted to the highest queue after a certain period. I have implemented this in a very similar fashion to the round-robin. However, instead of struct comp3520_rq having only one sched_comp3520_entity it has three of them. (I have made multi-level to be three levels) Inside this structure, it also contains a unique member called tick counter which is used to count the global tick of this run queue therefore later on notify the scheduler when to promote all the queues to the highest priority. I have also modified the sched_comp_3520 entity such that different queues should have different available tick counts decrementing the priority. This means that I have considered such that when an individual task runs for a certain amount of time then it should decrement its priority to another queue. These values are initialized as Marcos on comp3520.c. FIRST_ PRIO_ TICK_ COUNT, SECOND_ PRIO_ TICK_ COUNT, and for the third one as well.

2.1.2 Explain how does this scheduler compare to your round-robin scheduler

I have used runqlat to test the latency of the designed scheduler. First here is the result.

```
pid = 230
     usecs               : count     distribution
         0 -> 1          : 1995694  |****************************************|
         2 -> 3          : 221      |                                        |
         4 -> 7          : 124      |                                        |
         8 -> 15         : 73       |                                        |
        16 -> 31         : 46       |                                        |
        32 -> 63         : 21       |                                        |
        64 -> 127        : 4        |                                        |
```

Round-robin result

```
pid = 234
     usecs               : count     distribution
         0 -> 1          : 2022024  |****************************************|
         2 -> 3          : 223      |                                        |
         4 -> 7          : 131      |                                        |
         8 -> 15         : 80       |                                        |
        16 -> 31         : 58       |                                        |
        32 -> 63         : 16       |                                        |
```

Multilayer feedback queue result

Both show a magnificent result as the wait time for the CPU is mostly less than 1 microsecond. I was surprised to see how much of the two schedulers are similar to each other. I had to give a bit of thought to this and here is my analysis on why. The test that I was running was extremely CPU

intensive. The reason why they have such a similar result is that the tick count for the process to decrement its priority was similar to round-robin (only twice as long) since there are so many tasks in the queue and before they experience the latency due to decrements of their priority, they get promoted to the highest queue. This means on such a test which is CPU intensive and has a lot of tasks, these tasks are most likely that they will stay on the topmost priority no matter what. However, these results can be very different on how much I tune the priority tick count macros in comp3520.

2.2 Critique

2.2.1 Limitation of my scheduler

The limitation of my scheduler is that the performance can be very different on how I tune the macros of available tick counts for each queue. It can be targeted for the I/O intensive process or CPU Intensive tasks but if they coexist as most of the queues will have such events, it may not be fair to certain tasks. I also have to mention that there is also a slight overhead coming from promoting the queues and decrementing their priorities.

2.2.2 Compares to the CFS

I have used runqlat to test the latency of the CFS and multi-level feedback queue.

```
pid = 232
     usecs               : count     distribution
         0 -> 1          : 1752815  |****************************************|
         2 -> 3          : 46900    |*                                       |
         4 -> 7          : 7124     |                                        |
         8 -> 15         : 1773     |                                        |
        16 -> 31         : 697995   |***************                         |
        32 -> 63         : 27599    |                                        |
        64 -> 127        : 6376     |                                        |
       128 -> 255        : 9656     |                                        |
       256 -> 511        : 18302    |                                        |
       512 -> 1023       : 24653    |                                        |
      1024 -> 2047       : 39099    |                                        |
      2048 -> 4095       : 3979     |                                        |
      4096 -> 8191       : 337      |                                        |
      8192 -> 16383      : 83       |                                        |
     16384 -> 32767      : 7        |                                        |
```

CFS scheduler

```
pid = 234
     usecs               : count     distribution
         0 -> 1          : 2022024  |****************************************|
         2 -> 3          : 223      |                                        |
         4 -> 7          : 131      |                                        |
         8 -> 15         : 80       |                                        |
        16 -> 31         : 58       |                                        |
        32 -> 63         : 16       |                                        |
```

Multi-Level feedback queue

As you can tell CFS's latency the medium is around 4 -> 7 microseconds. Since it is considering the weight of the process which MLFQ can't, it is doing this to make the tasks get scheduled reasonably fair fashion. However, in this case, our Multi-level feedback queue is performing better since everything is on the same priority level due to the adjusted tick counts. However, this also suggests that it's not taking into account those processes that we need to put in a lower priority but putting it at the topmost means some of the processes might have starvation.