WRITEUP FOR ASSIGNMENT 2

By Arman Rajesh Ganjoo 2021018

Q1.1

I experimented with Linux thread scheduling in this question by creating 3 threads and giving them different scheduling policies (SCHED_OTHER, SCHED_RR, SCHED_FIFO) along with different priority levels (except for threadA using SCHED_OTHER) for each iteration (total 10 iterations for finding the variance in data). Each thread used the same logical function for computation. All of them counted from 1 to 2^32 asynchronously with each other.

For timing the threads, I used:

struct timespec x -> Found in <time.h> header

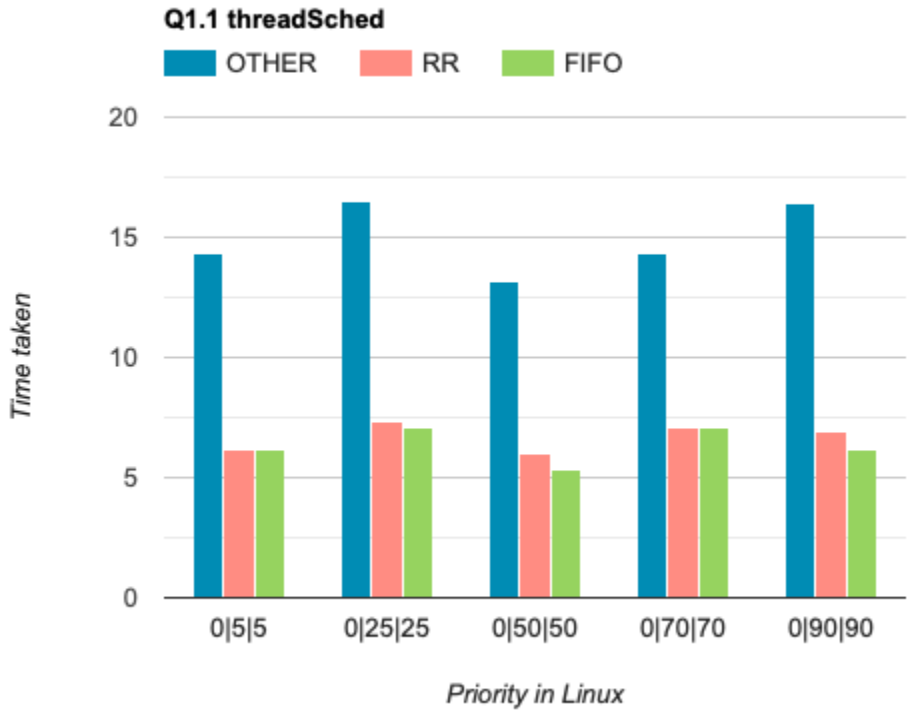
For modifying the scheduling policy and priority levels of the threads, I used:

(There are found in <sched.h> header)

```
pthread_attr_t attrThread;
struct sched_param tsp;
pthread_attr_init(args);
pthread_attr_setinheritedsched(args);
pthread_attr_setschedpolicy(args);
pthread_attr_setschedparam(args);
```

For each iteration, the policy of the threads remaining constant i.e ThreadA was SCHED_OTHER, ThreadB was SCHED_RR, ThreadC was SCHED_FIFO.
But the priority level of ThreadB and ThreadC kept on incrementing by a certain factor for which I used the formula -> SCHED_RR/FIFO_minPriority + iterationNumber * priorityBreak.

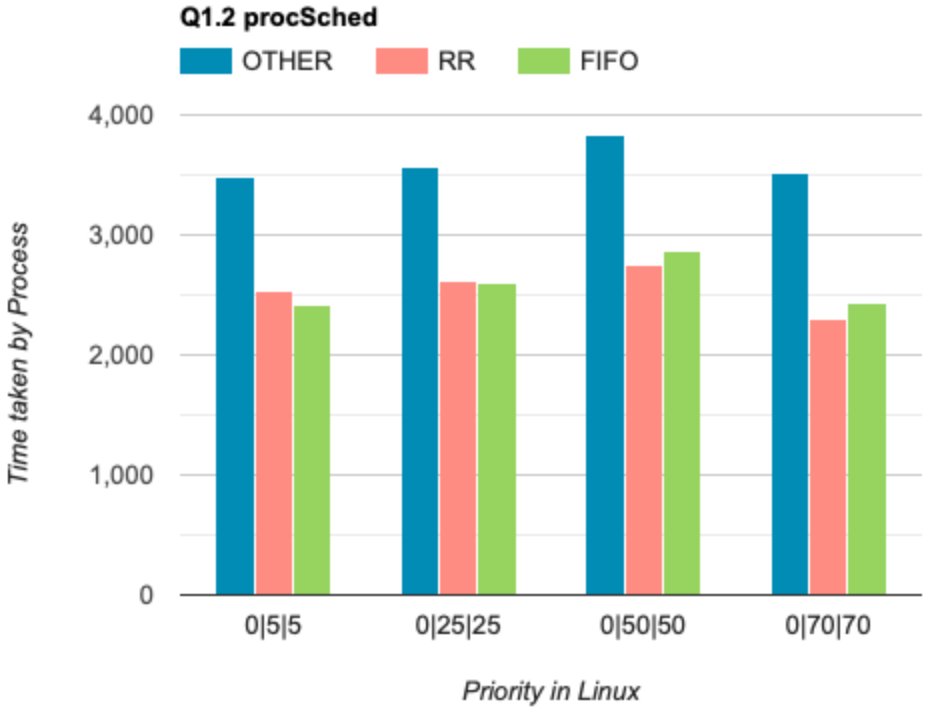
After the whole experimentation, it was clear that threads running with the scheduling policy of SCHED_RR and SCHED_FIFO were much faster than threads running with SCHED_OTHER. The fact that SCHED_RR and SCHED_FIFO policy threads could have higher priority levels also meant that the run-time of these particular threads could be decreased further with an increase in their priority levels.



Q1.2

In this question, I had to create 3 different processes using the fork() command found in the <unistd.h> header. These processes had different scheduling policies (found in the <sched.h> header) in the order -> proc1: SCHED_OTHER, proc2: SCHED_RR, proc3: SCHED_FIFO.

Each of these processes was supposed to run a bash script executable (which is created after running chmod +x ./scriptname on the bash script). The bash scripts contain the same content but are executed by their respective procs. The bash scripts were executed using execl() command. When these bash scripts were executed, they attempted to compile our custom kernel (which takes a long time!). All these 3 processes ran asynchronously and compiled the custom kernel.
The observations I observed were more or less the same as the ones I noted in Q1.1 where threads were used instead of processes.



Q2

In this particular question, I had to generate a patchfile which could be used to patch a kernel and add the custom syscall I made (kernel_2d_memcpy) to it. Before patching the file though, we had to add the custom made system file to the kernel. This syscall was made to copy a 2D matrix from one memory address to another - in gist, just clone the memory. But this cloning was supposed to be done using the __copy_from_user() and __copy_to_user() kernel functions. I had to include the <linux/kernel.h> header file to create this particular syscall.
After creating the custom syscall, I had to create a Makefile which compiled the file which contained the custom syscall.
After that, I had to add the custom syscall as a new entry to syscall_64.tbl (as my machine runs on x86_64 architecture). This .tbl file was in the kernel directory.
After that, I just compiled the kernel and it successfully added the custom and new syscall to my kernel.

To create the patchfile, I had to use `diff -ruN` command on 3 files:

1. The syscall_64.tbl
2. The kernel_2d_memcpy.c

3. The Makefile in the kernel directory

This created the patchfile which can be patched onto a kernel using the command `patch -p0 < patchfile`

Reference: https://brennan.io/2016/11/14/kernel-dev-ep3/