# Introduction

My code for the first question contains the following files: threadScheduling.c (Q1.1), processScheduling.c (Q1.2), a Makefile, and clean.sh. Since I wished to have the option to initiate threadScheduling and processScheduling separately, I chose not to run their `.out` files in the Makefile itself. Further, to clean up the remnants of past compilations, I made clean.sh which simply executes `rm -f *.out *.o.` As a general note, I realise that Nice values are inversely proportional to their priority as compared to other SCHED_OTHER processes/threads. I chose to take tests in ascending order of Nice (for SCHED_OTHER) and Priority (for realtime) values merely because I felt it would make the output I expected more apparent.

# 1.1. Thread Scheduling

threadScheduling.c works by initiating three threads using pthread_create() in the main function of the program. Each function is configured more or less in the same way:

1. The scheduling policy for the thread is defined using pthread_setschedparam().
    - in the case of Thread A (allotted SCHED_OTHER), it's nice value is then set using setpriority().
2. A message stating that the thread has been created is printed.
3. The start time is stored using clock_gettime() in a timespec structure called start.
4. The counting function is executed.
5. Once the thread has finished executing, clock_gettime stores the end time in timespec stop.
6. The duration is calculated and printed.
7. The function returns to main, where pthread_join() is making the parent process wait.

Outputs were taken by averaging three runs of each priority value.
The expected output was for the realtime processes (B & C) to always finish after A, which was indeed the general pattern. Between B and C, the thread which finished first was completely incidental. Further, my testing showed that changing Nice values didn't seem to make much of a difference to runtimes, which can be explained by the fact that there were no other user programs running, and so, in a scarcity of other programs competing for CPU time, our thread gets executed first. Similarly, changing the priority Value for B & C also yielded very little difference.

# 1.2. Process Scheduling

processScheduling.c has a single function (compile3())) which executes the fork and exec functions required for creating all three processes. The processes needed to be initiated at the same-ish time, and therefore I've used a sort of nested fork based system to create all three processes. Start time is taken before each (so it is available to the parent process). If the pid==0 (i.e. the current process is the child), sched_setscheduler is used to set the scheduler, and execvp is used to run a bash file corresponding to the process. Else, if pid>0, the process is the parent, and creation of the next process takes place and so on. At the end, a while loop makes the parent process wait for all three child processes to elapse, and stores the end time. Finally, the durations are calculated and printed. The bash files simply executes three separate (but same) copies of the kernel install files.

The results consistently show the realtime processes taking lesser time than the SCHED_OTHER process, and that too by a large margin. This is unsurprising, given that, as mentioned in my analysis of the results of 1.1, the priority for those (1-99) is always greater than SCHED_OTHER processes (0, and nice values are not comparable to the aforementioned priority), which are handled by the CFS.

## 2. Kernel Memory Copy

References:

1. https://brennan.io/2016/11/14/kernel-dev-ep3/
2. https://cool-dev.in/posts/How-to-Implement-a-System-Call-In-Linux/

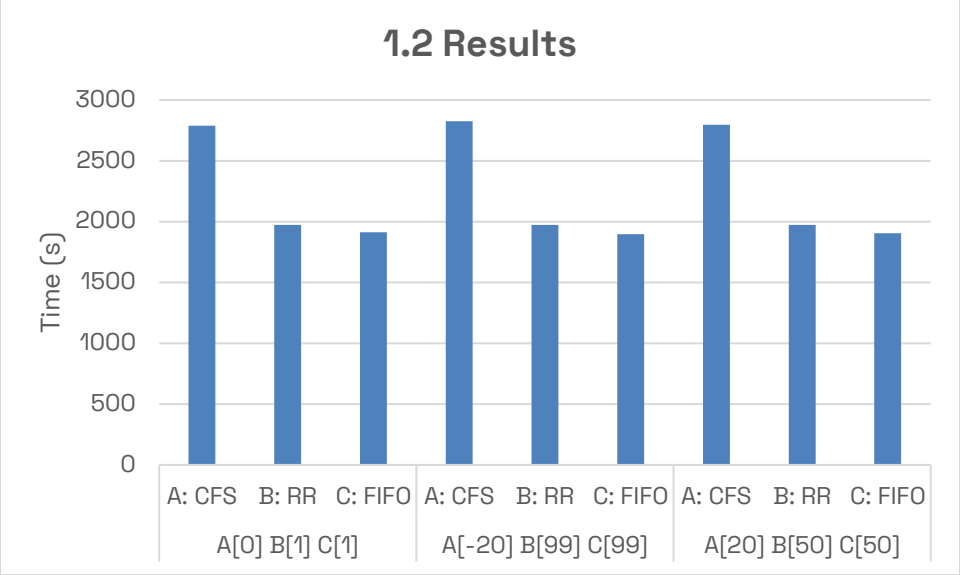
I've created my syscall by adding the code present in kernel_2d_memcpy.c into kernel/sys.c in the extracted kernel files. Further, I modified arch/x86/entry/syscalls/syscall_64.tbl to include an entry for my requited syscall.

The functionality for the syscall is simple: a buffer array is created in the kernel space, to which copy_from_user() copies the source array into, and copy_to_user() copies the buffer into the destination array.

testcall.c is also quite simple – it simply creates an array and compares it manually to the copied array, which is generated using syscall(syscall_number).

## Results


| A[0] B[1] C[1] | | | A[-20] B[25] C[25] | | | A[-10] B[50] C[50] | | | A[10] B[75] C[75] | | | A[19] B[99] C[99] | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A: CFS | B: RR | C: FIFO | A: CFS | B: RR | C: FIFO | A: CFS | B: RR | C: FIFO | A: CFS | B: RR | C: FIFO | A: CFS | B: RR | C: FIFO |
| 2.82 | 2.297 | 2.06166 | 3.8763 | 2.0673 | 2.1476 | 4.1366 | 2.0816 | 2.164 | 3.0983 | 2.260867 | 2.06733 | 3.14 | 2.31545 | 2.06962 |




| A[0] B[1] C[1] | | | A[-20] B[99] C[99] | | | A[20] B[50] C[50] | | |
|---|---|---|---|---|---|---|---|---|
| A: CFS | B: RR | C: FIFO | A: CFS | B: RR | C: FIFO | A: CFS | B: RR | C: FIFO |
| 2787.65 | 1974 | 1911.61 | 2824.147 | 1973.301 | 1897.775 | 2796.41 | 1973.72 | 1903.58 |