Project Development Environment and Main Goals

The main purpose of this project was to compile and analyze a custom virtual CPU object within the Linux kernel. We were required to assess various scheduling policies and apply them within our CPU. To conduct this project, it was necessary to gain an understanding of kernel programming, system calls, and the four scheduling algorithms of First-Come First-Serve (FCFS), Shortest Remaining Time First (SRTF), Round Robin (RR), and Priority scheduling. The virtual CPU object we created (ku_cpu) needed to be able to handle and demonstrate the basic implications of each scheduling policy.

The development environment we used was the Ubuntu 18.04.02 (64-bit) along with the Linux kernel version 4.20.11. The development and testing were done with a Virtual Machine environment; specifically, VirtualBox. This setup provided an appropriate platform for our modifications and tests of our scheduling policies. Overall, by using these tools for our project, it allowed us to simulate a real-world operating system which was valuable for conducting our programming tasks.

Explanation of CPU Scheduling and Policies

By creating a virtual CPU in kernel space, we were able to better analyze the scheduling policies we were taught in class. The CPU; “ku_cpu” was responsible for handling the scheduling of four different polices: First-Come, First-Served (FCFS), Shortest Remaining Time First (SRTF), Round Robin (RR), and Priority Scheduling (with pre-emption). Each policy would be tested to check its efficiency and behaviours.

FCFS operates by handling processes in the order they arrive, however its downside is that it could result in long wait times for longer processes. SRTF prioritizes the shorter tasks, however it could also result in long jobs never getting a chance to run if shorter jobs continually arrive. RR scheduling employs time slices and runs processes in a cyclic order; allowing each one equal time in the CPU (as governed by the specified time slice). The RR method is quite useful and fair, however the context switching overhead is one of its downsides. Lastly Priority scheduling runs processes based on priority, and higher prioritized tasks are able to pre-empt the lower ones. This could be useful for time sensitive tasks however similar to SRTF this could also lead to a starvation of tasks if higher priority tasks constantly get enqueued. However, there are several methods and tools that can assist to avoid the starvation of tasks such as aging or implementing specific rules or thresholds to stop these occurrences. Overall, each scheduling policy proved to have its own pros and cons associated with itself.

The user-space requests the CPU time and processes can specify parameters like job time, start delay, process name, priority, or policy. The virtual CPU manages requests, builds a waiting queue, and tracks which process is currently using the CPU. Through building our own CPU we were able to better understand the trade-offs in different scheduling methods and where their more appropriate use cases could be applied.

Implementation

The main implementations required for this project reside in the '*ku_cpu.c'* and the '*ku_cpu_user.c*' files. The goal was to create a user space which would process requests to our CPU, use a system call to request the CPU, and print out the response times and wait times for our processes. We also created a kernel space that would implement our custom CPU, use a system call handler, and demonstrate the four different scheduling policies.

In '*ku_cpu_user.c*' it would receive prompts from a *run file* that would pass parameters to the user space like Job Duration (in seconds), Starting Delay (in seconds), Process Name, and Priority (only for priority scheduling). An example command in "run" would be: **./ku_cpu_user 5 1 B &** for FCFS/SRTF/RR or **./ku_cpu_user 5 1 B 1 &** for Priority scheduling which requires an extra parameter. This command would imply that process "B" will request CPU time after 1 second and will use the CPU for 5 seconds, and if priority was necessary the additional argument would denote a priority of 1 in this example. Most of the user code was built around the example code in the report along with new modifications to include response time and changes made to accommodate the different scheduling policies. The user code files will be attached in the submission. The FCFS, SRTF, and RR policies could be ran with the same user space code and the Priority scheduling used a different user space code since it required an additional argument.

In '*ku_cpu.c*' the main concepts of the scheduling policies were implemented. These files were mainly based of the example code skeleton provided in the assignment outline. However, four separate code files were used for each of the policies. Most of these files were very similar to each other and used the same helper functions however the logic inside the system call was modified each time to accommodate each different scheduling policy. And sometimes a few additional helper functions were added if they would better accommodate a specific policy. Essential data structures included a Waiting queue for processes, a Variable to track the current process occupying CPU (called "now"), and a PID assigned to each process.

In summary; FCFS would just use a basic array and work jobs in the order that they arrive. SRTF used the FCFS code as a base but it also used an additional helper function of "organize_queue()" which would reorganize the queue and always put the shortest job first; if

the “now” variable was still the first in line it was worked otherwise a “Turnover ---->” was used and the shorter job was switched to “now” and the longer job went to the back of the line. The RR implementation used a counter called “time_spent” to keep track of how long the job had been worked and was compared to with the time slice threshold. If it was at the threshold and had time remaining it would be paused and move to the back of the line to give another job a turn, but if it wasn’t it would still be worked or maybe finished if the job time reached zero. The priority queue worked very similar to the SRTF code however it used a helper function called “is_highest_priority()”. Before allowing a job to work it would first check to make sure it still currently had the highest priority present. If so, it would be worked, and if not, it would receive a “Reschedule ---->” statement and be moved behind jobs with higher priority and “now” would be changed to work the highest priority job present. The kernel code files will be attached in the submission for further clarity.

Experiment Results and Logs

The below data illustrates the response times and wait times recorded from processes A, B, and C under different scheduling policies (FCFS, SRTF, RR, and Priority Scheduling). Note sometimes results varied +/- 1 sec between each run, but that was an expected outcome for this experiment. Each policy had its own pros and cons observed from the tests. SRTF usually had lower times for both times while FCFS usually has higher times for both. RR showed to have the highest wait times of them all. While SRTF and Priority had the lowest (or best) wait times. Overall, the choice of scheduling policy largely impacted both the response and wait times.

RESPONSE TIMES (sec)					WAIT TIMES (sec)				
Process	**FCFS**	**SRTF**	**RR**	**Priority**	**Process**	**FCFS**	**SRTF**	**RR**	**Priority**
A	0	0	1	0	**A**	0	0	7	0
B	6	0	0	0	**B**	6	3	9	5
C	11	0	0	10	**C**	10	8	11	10

Below is a compilation of the PROs and CONs determined whilst analyzing the four policies

POLICY	PROS	CONS	GOOD USE CASES
FCFS	- Simplistic, straight forward to implement - Fair in relation to FCFS	- Can result the "convoy effect": short processes wait for long ones - Poor performance sometimes with average wait time and response time - Not appropriate for time-sharing systems	- Good for batch systems or in cases where processes are of similar lengths
SRTF	- Can decrease average wait and response time - Relatively more efficient	- Requires accurate estimate of length of processes (not always possible to know) - Can sometimes lead to starvation of longer processes (might need aging or thresholds)	- Good for systems where process lengths are able to be predicted - Good for real-time systems
RR	- Fair allocation of CPU time for each process - Can help preventing starvation - Good for time-sharing	- Context switching can add further overhead to the system - The performance could get better or worse; depending the choice of the length of the time slice	- Good for time-sharing or if response time is critical - Good for environments with many short processes
Priority	- Prioritizes important tasks, that way critical ones can be done first - Can add pre-emption (like in this project) to improve responsiveness	- Can sometimes cause starvation of lower priority tasks (could need aging to thresholds to combat this) - It requires an appropriate standard or policy to set priorities (can't just let everything be top priority, need a system to define)	- Best for systems where some tasks are more important than others - Good for real-time systems with certain high priority jobs

Problems Encountered and Applied Solutions

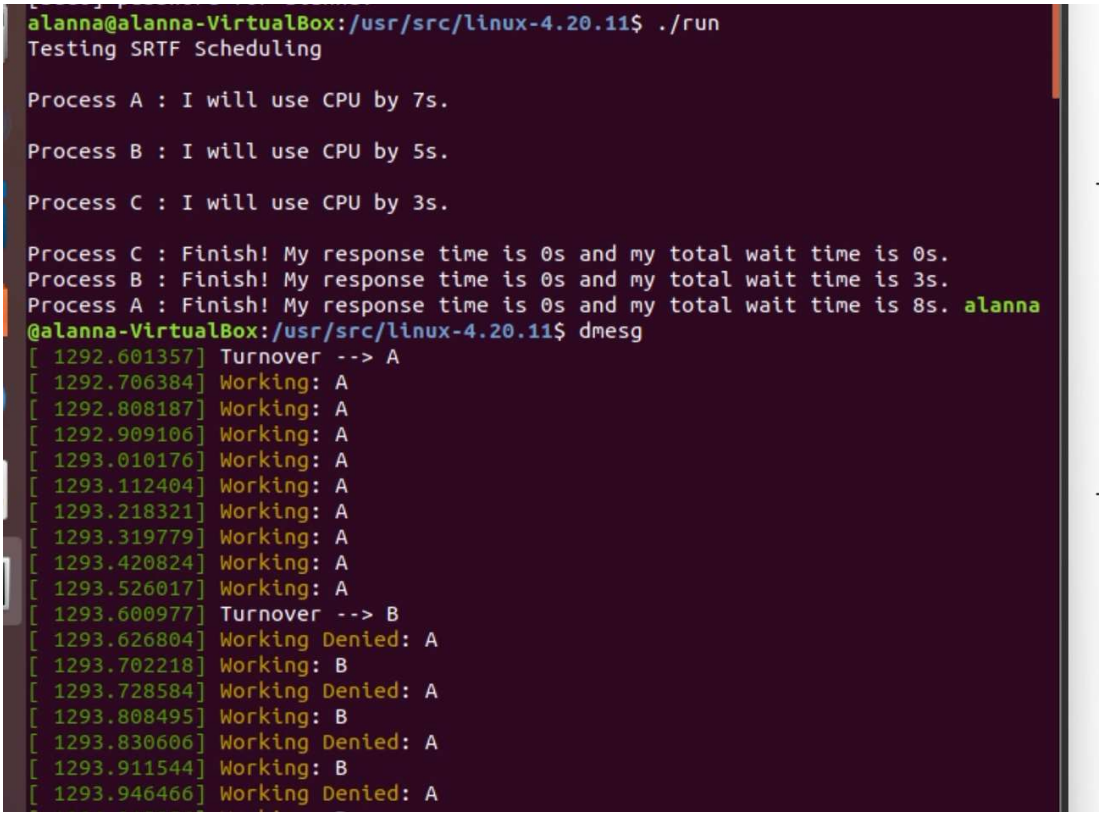
One main issue faced was encountering interferences coming from my previous project, which caused my code to run infinitely and print my incorrect outputs. After some investigation I found some error statements in my make and make install scripts. After deleting my project 1 related works, my project 2 work seemed to perform much better. I think perhaps some similar variable names or declarations maybe have been confusing my system. Additionally, I also had to ensure correct declarations in syscalls.h file and syscalls_64.tbl files. After this I learnt to pay closer attention to the outputs of make and make install commands. These commands often highlighted critical problems which were sometimes the difference between my code and desired outputs either working or not working.

Another challenge I encountered again (since last project) was the lengthy processing times associated with make, make install, and reboot commands. To mitigate this, I again deciding to use commands like sudo make -j8 or -j12 to speed up the process. Additionally, I started coding more in VS code and then copying and pasting my work into the VM. This helped for better readability for me and for better functionality for me as I am more accustomed to this environment for coding in. This helped me to better implement my works in this project.

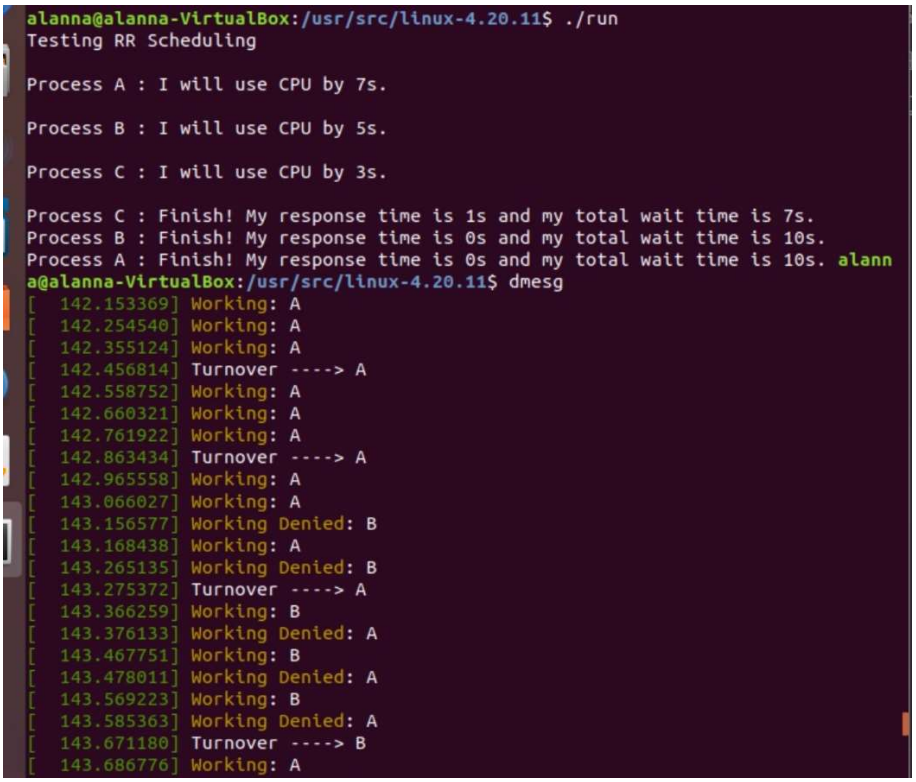
Snapshots

Below are the results after running for First-Come, First-Serve (FCFS) Scheduling:

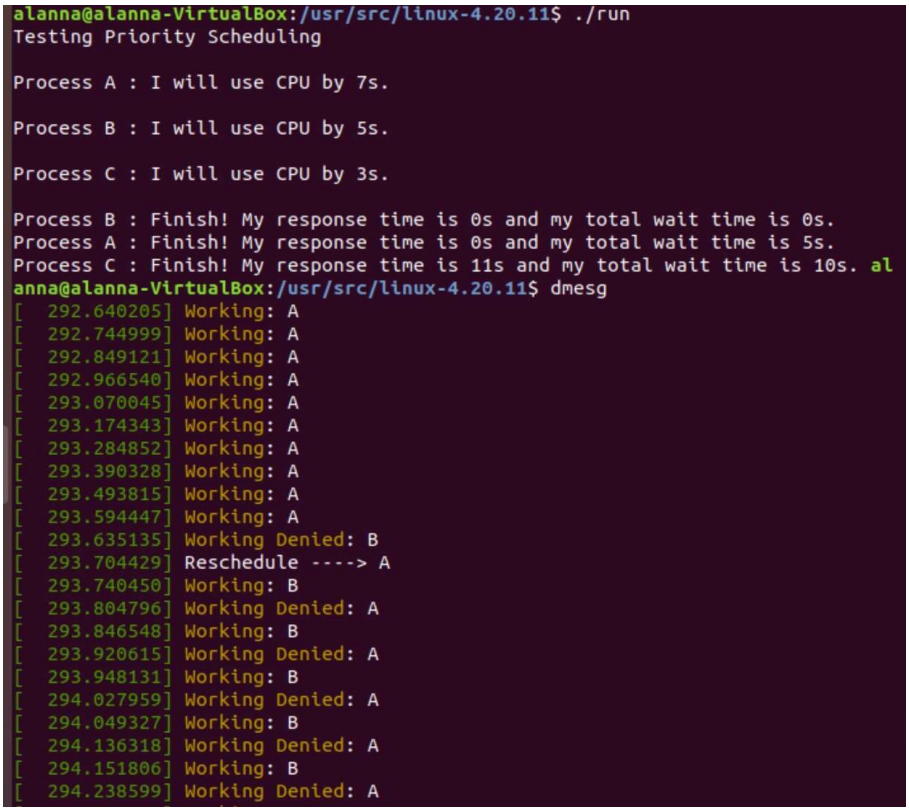
Below are the results after running for Shortest Remaining Time First (SRTF) Scheduling:



Below are the results after running for Round Robin (RR) Scheduling:



Below are the results after running Priority (with pre-emption) Scheduling:



Note: Full results text files can be found in the attached submission packet as .txt files.