Austin Dailey
Nikolai Downs
CS 453
Prof. Nico
2/17/2025
Lab 4

Introduction:
In this lab we modified the kernel of Minix3 to print the @ symbol anytime the CPU is idle.

1. **Find the correct location for a print statement**

Approach:

Our first task with this assignment was to find a place that has the knowledge of all current tasks. Our first instinct was to find the place where tasks are scheduled, similar to LWP. We figured there may be a spot in the code that like LWP, had a scheduler that would grab the next process in the queue. If there were no new tasks in the queue, then we could assume the system was idle. We first looked through main.c to get a general idea of the system. In this file, we did find some code that referenced scheduling.

The next location we looked at was system.c, we were unsure of what would all be in here, but given that scheduling processes is fairly important to the system, we thought it would be a good place to look. At first, we thought we had struck out when we saw the /*Process Management*/ section just mapped some signals to functions however at the very bottom we found the sched_proc function. We thought with this function we would be able to find a spot to put the print statement that would print if there were no schedule processes.

Finally, we decided to look in proc.c, since we thought we may be able to find a schedule_proc call. We ended up finding something even better than schedule_proc, an IDLE function. The description of this function told us it was exactly where we wanted to put the print statement as it was called whenever there was no work to do. We still wanted to know where this call came from and we found it in switch_to_user() which we also discovered is where processes are queued. The function we found earlier in the main must have just been used for boot tasks, as this function used enqueue() directly.

Problems Encountered:

1. When looking through main.c, we found some code that dealt with scheduling processes. However, after reading it in more detail, it seemed to be more of a one-time run, and specifically the line `for (i=0; i < NR_BOOT_PROCS; ++i)` told us that this was probably a one-time loop that ran all the root, kernel, and user boot tasks. So we must continue looking.

2. We realized the schedule_proc functions usage was to be called with a function that needed to enter the queue, so putting the print statement within would not work. We tried to find where this function was called but were unsuccessfully so we kept looking.

Solutions:
The solution was to put the print statement in the IDLE function or the switch_to_user function and not in main.c or system.c

Lessons Learned:
The lessons we learned included that the idea of a process scheduler is very common, so looking for something that emulated that was a good idea. We also learned that in Minix, there are separate locations and code to schedule boot processes and user processes which explained the one-time scheduling that was happening in main.

2. Build the modified kernel

Approach: Before doing anything to any file in the kernel folder we decided to make a tar backup of it using **tar xvf kernel_backup.tar kernel** in the /usr/src directory because getting it right on the first try would require a miracle. We got this backup and copied over to WSL for safekeeping. To do our reading of the code and modifying we copied the kernel directory to a local WSL instance so that it could be viewed and modified in VSC. Since by definition once you are in the idle task there is nothing to do, we simply added an if (1) { printf("@\b");} statement. We then needed to compile the kernel, install it, and reboot it. Within the /usr/src/tools directory we ran **make hdboot** and ran into some issues. Once those issues were fixed we were able to install it, reboot, and see the @ symbol being printed.

Problems Encountered:
1. The main problem we encountered were continuous issues with the Makefile in /usr/src/kernel. We kept getting error messages like" set: Illegal option - *** Error code 2" and "make: "/usr/src/kernel/Makefile" line 10: Unassociated shell command "watchdog.c"
2. When trying to fix these errors, we did some research and found that the syntax was possibly bad due to our modifying the file in VSC which uses Windows-style line endings among other things. Our first attempt to fix this was to use a program called dos2unix which we installed on wsl, ran on all the kernel source files, and recopied them over to the minix machine. This fixed some errors but not all of them. Since we had made such a small adjustment to the code we knew it was likely still the coping to windows and back process that had messed something up.

Solution:
Ultimately our solution was to restart with a fresh kernel directory. Since we had backed up the kernel folder we simply ran **tar xvf kernel_backup.tar** and now we had the original source files again. Instead of modifying the file in VSC we just used VI to modify it now that we knew where it needed to go. After this refresh the kernel compiled correctly and we were able to install it.

Lessons Learned:

We were reminded that Linux and Windows-based systems have many subtle differences that cause incompatibility. We also were reminded that things like line endings, and tabs vs spaces all have effects in a makefile since they use indentation-based syntax.

3. **Devise a test**

Approach:

Now that we had successfully compiled and installed the kernel we rebooted the system and found the @ symbol being printed almost always. Our first idea to test it by copying a big file from Minix to WSL to see if it disappeared during the transfer however when douing this we only saw the @ flash for a split second and reappear. Our next test was to just run and infinite loop to make sure it was disappearing when something was happening by running **while true; do :; done;** and indeed the @ symbol was no longer printed until we killed that process.

Problems encountered:

1. At first, we were unsure of why the @ symbol didnt seem to be disappearing during an SCP call, however after doing some research we realized most of the time taken during a big file transfer is dealing with I/O. Since SCP is mostly an I/O-bound task it doesn't actually use the CPU much and therefore still leaves it idle.

Solution:

Instead of using the file transfer as a test, we decided to use a process that would guarantee CPU usage like an infinite while loop.

Lessons Learned:

We learned that SCP is not a good metric of CPU usage since it is mostly an I/O bound process.