Christopher Wycoff (951078512)

# CIS 415 Operationg Systems

## Project 2 Report

Submitted to :
Prof. Allen Malony, Jared Hall, Grayson Guan

Author:
Christopher Wycoff

**Introduction**

Parts 1-4 build up a program that creates a round robin execution scheduler.

- Part1: Forks processes, executes other programs and waits for them to finish.
- Part2: Takes processes from part 1, sends signals to these processes to stop, and start back up.
- Part3: Uses processes from part 1 and 2. Strategically sends signals to processes to start/stop in a round robin scheduling fashion.
- Part4: Expands upon Part3 by reading process information from the /proc directory. This allows for more informative output to be printed.

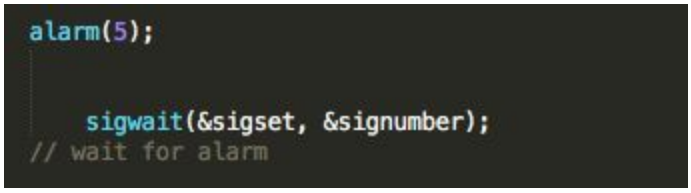
**Background**

Process scheduling is a fundamental OS concept. The round robin scheduling algorithm, used in this project, is a building block for many OSs. Signaling is key in this project, signaling is a form of IPC. Signaling interrupts the normal flow of a process' execution, thus allowing us to start and stop processes in a strategic manner.

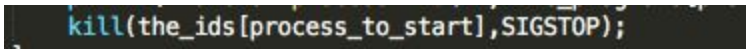
**Implementation**

In this project we explore how to write a .c program that implements a rudementery round robin scheduling algorithm.

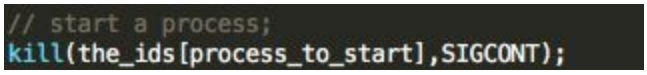
Here are some snapshots of the code:



Process is stopped for 5 seconds.



process is stopped



continued

```
////////////////////////////////////Begin Proc Read/////////////////////////////////////////////
            proc = opendir("/proc");
            if(proc == NULL) {
                perror("opendir(/proc)");
                return 1;
            }

            while((ent = readdir(proc))) {
                if(!isdigit(*ent->d_name))
                    continue;

                tgid = strtol(ent->d_name, NULL, 10);
                for (int fork_iterator = 0; fork_iterator < number_of_programs; fork_iterator++ ){
                        if (tgid == the_ids[fork_iterator]){
                            printf("\n");
                            printf("Program %s \n" , the_programs[fork_iterator]);
                            display_status(tgid);
                            get_exec_time(tgid);
                            display_memory(tgid);
                            display_io_write(tgid);
                            display_io_read(tgid);
                            printf("\n");
                        }
                }

            }

            closedir(proc);
////////////////////////////////////////////////////////////////////////////////////////////////
```

Read from /proc

```
Resuming child process: 3773
Name of process: ./iobound

Program ./iobound
  3773 R (running)
Time executing on cpu (miliseconds):  4863.005898
Memory used: 4372 kB
IO (write):  4917710000
IO (read):  1948


Program ./cpubound
  3774 T (stopped)
Time executing on cpu (miliseconds):  0.215446
Memory used: 4512 kB
IO (write):  0
IO (read):  0
```

Output while scheduling programs

**Performance Results and Discussion**

Per Grason's suggestion I used the return value from waitpid() for checking if a process is terminated. Does not cause issues from the testing that I have done. I tried to use the

WIFEXITED() logic with status, but this did not work, I tried several permutations and this failed, thus I left it as it was.

As far as I can tell it works as specified. If there are any issues with edge cases I apologize. There were one or two times during the building of the program where an error was fixed by restarting the vm. If there are any issues with the scheduling logic while assessing, I recommend restarting the VM.

**Conclusions**

I learned about:

- Implementing signals between processes
- The nature of and how to read the /proc folder
- How the exec() system call behaves
- How to fork processes in c
- How to stop and start processes in c