CIS 415 Operating Systems

Project <2> Report Collection

Submitted to:
Prof. Allen Malony

Author:
Sophia Zhang

951870312
sophiaz

Report

Introduction

This project involved the implementation of the round robin scheduler for managing processes in a simulated environment. The goal of this project is to understand how a scheduler manages multiple processes by alternating CPU time among them in a cyclic manner. To achieve this, the baseline of the project involved building each step from scratch and adding functionalities to the shell.

Background

The round robin implementation is a scheduling algorithm that assigns a fixed amount of quantum time to a process before moving on to the next. Processes that have not been completed yet will be put in at the end of the queue until terminated. This ensures fairness by allowing all processes to share CPU time equally to prevent starvation. In this particular project, we were given two files called `io bound` *and* `cpu bound` *to use. We called these two file functions and set a timer for 10 seconds on it to mimic that round robin effect of switching processes.*

To implement this, signals were a huge part of the project. Sending signals to the child process to have it execute or wait until it receives a signal with `SIGSTOP`, `SIGCONT`, *and* `SIGALRM`. *The* `SIGSTOP` *was used to stop ongoing process so it can move onto the next one.* `SIGCONT` *was used to continue the next process (the one after sigstop). And following would be an alarm that would be triggered every 1 second. The lecture about synchronization really helped me understood this process of sending signals using* `wait()` *and* `signal()`. *It explains how these functions work together to manage the execution flow between processes, which is something very similar to what we are implementing here.*

Implementation

Part one of the project involved reading from an input file that consisted of commands to run and parsing the results into an args array where `args[0]` *was the command, and the rest of the array was flags and arguments that went along with it.*

```
----------------------------------------
Command(args[0]) : ls
args[1]: -a
args[2]: -r
args[3]: -s
----------------------------------------
Command(args[0]) : sleep
args[1]: 1
----------------------------------------
Command(args[0]) : invalid
args[1]: name
----------------------------------------
Command(args[0]) : ./iobound
args[1]: -seconds
args[2]: 10
----------------------------------------
Command(args[0]) : ./cpubound
args[1]: -seconds
args[2]: 10
----------------------------------------
```

Finally, we had to fork each process for each command and had the parent process wait for each process to finish. We used the array `pid_array` *to keep track of how many processes were created.*

In part two of the project, we introduced signal sending to processes. I used `SIGSTOP` *and* `SIGCONT` *to control*

the execution of the process by pausing and resuming them at designated times, in this case `sleep(3)`*. Effectively, this simulating a form of scheduling where the parent process manages the execution flow of a child process in a controlled manner.*

In part three of the project, we built upon the concepts from Part 2 by implementing a signal handler. The signal handler sends `SIGCONT` *signals to the child processes, which are initially blocked. These child processes wait for the* `SIGCONT` *signal to indicate that it's their turn to use the CPU. This behavior was set up by creating a signal set (*`sigset`*), where I added* `SIGCONT` *and then blocked it using* `sigprocmask`*.*

```
sigemptyset(&sigset);
// Add CONT to the signal set
sigaddset(&sigset, SIGCONT);
// Block SIGALRM signal
sigprocmask(SIG_BLOCK, &sigset, NULL);
```

By doing this, the child processes created thereafter would be placed in a blocked state, waiting for the `SIGCONT` *signal from the parent process to resume execution. Once a child process receives the* `SIGCONT` *signal, it runs for 1 second (this was implemented using* `alarm(1)`*) before being stopped by the parent process with the kill command. The round-robin scheduling algorithm then selects the next process in the pid_array.*

```
/* Round Robin Scheduler: Switches to Next Process Each Time Slice */
void round_robin(int sig) {
    // Stop the currently running process, if active
    if (pid_array[current_process] != -1) {
        kill(pid_array[current_process], SIGSTOP);
    }

    // Move to the next process
    current_process = (current_process + 1) % pids;
    while (pid_array[current_process] == -1) { // Skip terminated processes
        current_process = (current_process + 1) % pids;
    }

    // // Continue the next process
    kill(pid_array[current_process], SIGCONT);
    printf("Scheduling Process: PID: %d\n", pid_array[current_process]);

    // Set the alarm for the next time slice
    alarm(1);
}
```

If a process has completed its cycle, I use `WIFEXITED()` *to check the status returned by waitpid(), and mark that specific process as -1 in the* `pid_array` *to indicate that it has finished. Passing into WIFEXITED() was the result status of* `waitpid()` *so if status is true, then finally we would mark the process as terminated.*

```
int status;
pid_t result = waitpid(pid_array[i], &status, 0);
```

In part four of the project, we were suppose to grab process data from the `/proc` *file, which is a mechanism that allows the kernel to reveal information about the system and the running processes. It contains system information such as pid, process state, CPU usage, etc. Access to this file required me to create a path using* `snprintf` *to grab the path to the proc file specific to that process given the* `pid`*. Then every would be grabbed and stored in a buffer using* `fgets`*. The information in the* `/proc` *file is hard to read. But the man page would clearly state what each number is and what type they are. Knowing so allowed me to use* `sscanf` *to parse*

```
sscanf(buf, "%d %*s %*c %*d %*d %*d %*d %*d %*u %*lu %*lu %*lu %*lu %lu %lu %*ld %*ld %*ld %ld %*ld %*ld %*llu %lu",
   &process_pid, &utime, &stime, &nice, &virt_mem);
```

the data and skip the ones I did not need. I'm not sure if there is any easier way of doing this but skipping each data value and counting them manually was time consuming.

Finally, the parsed values are stored in the designated variables I created for them, where later on I converted the time to second using `sysconf(_SC_CLK_TCK)`*. This allowed* `utime`*,* `stime`*, and total time to be printed out in seconds instead.*

In part 5, I implemented a adjustable scheduler to optimize CPU utilization by dynamically allocating time slices based on process behavior. The scheduler itself distinguishes between cpu bound and i/o bound processes by analyzing their runtime patterns. CPU bound required more CPU usage time so it was given a longer time slice, while i/o bound had shorter cpu usage time hence shorter time slice. This adjustment was aim to improve efficiency that CPU cycles are not in busy waiting while i/o bound processes are running. This was implemented using a time slicing array, where it got updated accordingly to increase or decrease time. This gets used as the time quantum instead of the default 1.

Performance Results and Discussion

I believe my codes runs well with the required requirements. It follows the round robin scheduling algorithm and alternates the cycle of processes. Below is an output sample of my part3 which includes the process creation and signaling.

```
sophie@debian:~/CS415/MCP:Ghost-in-the-shell$ ./part3 -f input.txt
Scheduling Process: PID: 46675
Scheduling Process: PID: 46676
Execvp Failed: No such file or directory
Scheduling Process: PID: 46677
Process: 46677 - Begining to write to file.
Scheduling Process: PID: 46678
Process: 46678 - Begining calculation.
Scheduling Process: PID: 46674
total 200
 8 part5.c   8 part3.c   4 part1.c    20 iobound      4 arch64_example_executables
20 part5    20 part3    20 part1       4 input.txt    4 ..
 8 part4.c   8 part2.c   4 Makefile     4 cpubound.c   4 .
20 part4    20 part2     4 iobound.c   16 cpubound
Scheduling Process: PID: 46675
Scheduling Process: PID: 46677
Scheduling Process: PID: 46678
Scheduling Process: PID: 46677
Scheduling Process: PID: 46678
Scheduling Process: PID: 46677
Scheduling Process: PID: 46678
Scheduling Process: PID: 46677
Scheduling Process: PID: 46678
Scheduling Process: PID: 46677
Scheduling Process: PID: 46678
Scheduling Process: PID: 46677
Scheduling Process: PID: 46678
Scheduling Process: PID: 46677
Scheduling Process: PID: 46678
Scheduling Process: PID: 46677
Scheduling Process: PID: 46678
Scheduling Process: PID: 46677
Scheduling Process: PID: 46678
Scheduling Process: PID: 46677
Process: 46677 - Finished.
Scheduling Process: PID: 46678
Process: 46678 - Finished.
All processes have completed.
```

If we take a closer look, it schedules a total of 5 processes, 1 of them failed which is expected because of an invalid command. The next two processes are ran and the results are shown, one of them is the `ls` *command which list out the files in the directory. The other command was the* `sleep()` *command, but we can not see that here. Following by the two other processes which we can see are alternating between each other. This is important because that is the whole idea of round robin, in this case we have io bound processes and cpu bound processes switching off each other.*

Next I have the output results from my `/proc` *file. Not all of the data is captured in the image, but after each full cycle. It prints out a table of information about each process that ran in that cycle.*

```
PID     utime   stime   time    nice    virt mem
48092 0.000000 0.000000 0.000000   0   2519040
48093 0.000000 0.000000 0.000000   0   5595136
48094 0.000000 0.000000 0.000000   0   0
48095 0.930000 0.060000 0.990000   0   2519040
48096 0.000000 0.000000 0.000000   0   2519040
Process: 48096 - Begining calculation.
total 200
 8 part5.c   8 part3.c   4 part1.c    20 iobound      4 arch64_example_executables
20 part5    20 part3    20 part1       4 input.txt    4 ..
 8 part4.c   8 part2.c   4 Makefile     4 cpubound.c   4 .
20 part4    20 part2     4 iobound.c  16 cpubound

PID     utime   stime   time    nice    virt mem
48093 0.000000 0.000000 0.000000   0   5595136
48094 0.000000 0.000000 0.000000   0   0
48095 0.930000 0.060000 0.990000   0   2519040
48096 0.720000 0.270000 0.990000   0   2519040

PID     utime   stime   time    nice    virt mem
48095 1.860000 0.130000 1.990000   0   2519040
48096 0.720000 0.270000 0.990000   0   2519040

PID     utime   stime   time    nice    virt mem
48095 2.780000 0.210000 2.990000   0   2519040
48096 1.360000 0.630000 1.990000   0   2519040

PID     utime   stime   time    nice    virt mem
48095 3.700000 0.290000 3.990000   0   2519040
48096 2.030000 0.960000 2.990000   0   2519040

PID     utime   stime   time    nice    virt mem
48095 4.620000 0.370000 4.990000   0   2519040
48096 2.710000 1.280000 3.990000   0   2519040

PID     utime   stime   time    nice    virt mem
48095 5.560000 0.430000 5.990000   0   2519040
48096 3.350000 1.630000 4.980000   0   2519040

PID     utime   stime   time    nice    virt mem
48095 6.440000 0.550000 6.990000   0   2519040
48096 4.040000 1.950000 5.990000   0   2519040

PID     utime   stime   time    nice    virt mem
48095 7.340000 0.650000 7.990000   0   2519040
48096 4.660000 2.330000 6.990000   0   2519040

PID     utime   stime   time    nice    virt mem
```

Lastly a report of part 5 where alarm time is based on type of execution, whether it is IO or CPU bounded. Where IO will have a minimum of 1, and CPU will as much as needed.

```
PID     utime   stime   time    nice    virt mem timeslice  type
60267 0.000000 0.000000 0.000000   0   2519040  1.500000 CPU Bound
60268 0.000000 0.000000 0.000000   0   5595136  2.000000 CPU Bound
60269 0.000000 0.000000 0.000000   0   2519040  2.500000 CPU Bound
60270 0.000000 0.000000 0.000000   0   2519040  3.000000 CPU Bound
60271 0.000000 0.000000 0.000000   0   2519040  3.500000 CPU Bound
Execvp Failed: No such file or directory
Process: 60270 - Begining to write to file.

PID     utime   stime   time    nice    virt mem timeslice  type
60267 0.000000 0.000000 0.000000   0   2519040  1.500000 CPU Bound
60268 0.000000 0.000000 0.000000   0   5595136  2.000000 CPU Bound
60269 0.000000 0.000000 0.000000   0   0  2.500000 CPU Bound
60270 0.910000 0.080000 0.990000   0   2519040  2.000000 I/O Bound
60271 0.000000 0.000000 0.000000   0   2519040  2.500000 CPU Bound
```

Conclusion

I found this project very interesting actually. Seeing and implementing something so close to what I use everything is really cool. And I learned quite a lot in this project as well, mainly how signal sending work and accessing the `/proc` file. Overall, this project was really interesting to implement, and I enjoyed the learning outcome.