[2020 OS] Project 1 Report

B07902084 資工二 鄭益昀

Background

This project is written for Prof. Chih-Wen Hsueh's 2020 Spring Operating Systems class by B07902084 資工二 鄭益昀.

Link to the girhub repository: https://github.com/ycheng627/OS_project1

Setup

- Host Operating System: Windows 10
- Hypervisor: Oracle VM Virtualbox version 6.1.4
- Linux Distribution: Ubuntu 16.04
- Compiled Kernel Version: Linux 4.14.25

Design Philosophy

Process Switching

- The code for switching between processes can be found in the files `process.h` and `process.c`.
- The definition of a process is as below:

```
struct process {
 char name[32];
 int ready_time;
 int exec_time;
 pid_t pid;
        int active;
        int exist;
};
```

- There are four main functions implemented:
  - setCPU - sets the affinity of a process to a particular CPU. I have set aside two CPUs, `SCHEDULER_CPU = 0` and `FORK_CPU = 1`. All new processes are assigned to their respective CPU.
  - maxPriority - takes in a `pid` and sets the priority of the pid to 99 by calling the function: `sched_setscheduler(pid, SCHED_FIFO, &parameter)`. Notably, `sched_setscheduler()` actually sets the priority for a thread, but since we didn't make any multithreaded program, each forked process can be treated as its own thread.
  - minPriority - same implementation as maxPriority, except with priority = 0
  - terminateProcess - takes in a `pid`, uses the `waitpid()` function to wait for the child, and set the process active and exist to 0.
  - childProcess - forks a process.
    - For the parent, set the process' pid, exist = 1, and minpriority
    - For the child, set the cpu and minpriority. Record start time, run for designated time, record end time, then print using system call

```
setCPU(pid, cpu){
    sched_setaffinity(pid, cpu);
    #CPU can take on 0 or 1 depending on SCHEDULER_CPU / FORK_CPU
}

maxPriority(pid){
    sched_setscheduler(pid, SCHED_FIFO, 99)
}

minPriority(pid){
    sched_setscheduler(pid, SCHED_FIFO, 0)
}

terminateProcess(proc){
    waitpid(proc->pid, NULL, 0)
    proc->active = 0
    proc->exist = 0
}
```

FIFO

- The FIFO scheduling policy runs the processes by their ready time. No preemption is done in FIFO.
- The pseudocode for FIFO is as below:

```
FIFO{
    qsort() //sort by ready time
    init() //set the cpu, and proc states
    while(1){
        if(running process is done running){
            terminate process
            running process id ++ //for FIFO, it must be sequential
            if (all process done){
                exit(0)
            }
        }
        if(another process can be spawned){
            spawn a new process
        }
        if(process is selected){
            process = active
            process = max priority
        }
        unit time()
        process exec time --
    }
}

```

RR

- The Round Robin scheduling policy runs the processes one by one, and preemptively switch the process out once it reaches the time limit of one round (500 units)
- A queue is used in Round Robin, such that each time a process is switched out, it is added to the end of the queue and a new process is taken from the start
- The pseudocode for RR is as below:

```
RR{
    qsort() //sort by ready time
    init() //set the cpu, and proc states
    q = createQueue()
    running_process = NULL
    while(1){
        running_process = q->front->proc
        if(another process can be spawned){
            spawn a new process
            add to end of queue
        }
        if(running process is done running){
            dequeue()
            terminate process
            if (all process done){
                exit(0)
            }
        }
        if(switch_time is up){
            tmp = dequeue()
            enqueue(tmp) //move the process to the end of queue
            switch_time = 500
            running_process = NULL
        }
        if(process is selected){
            process = active
            process = max priority
        }
        unit time()
        process exec time --
        switch_time --
    }
}

```

SJF

- The Shortest Job First policy runs the shortest of the currently available jobs. It does not get preempted.
- A priority queue is used in SJF, the priority is set based on the execution time remaining (lowest first)
    - Different from RR, the current running job is NOT in the priority queue, this is to prevent a new job from pushing it downwards
- The pseudocode for SJF is as below:

```
SJF{
    qsort() //sort by ready time
    init() //set the cpu, and proc states
    pq = createPriorityQueue()
    running_process = NULL
    while(1){
        if(running process is done running){
            terminate process
            if (all process done){
                exit(0)
            }
        }
        if(another process can be spawned){
            spawn a new process
            add to priority queue
        }
        if(no process is running now){
            running_process = pq.pop()
        }
        unit time()
        process exec time --
    }
}

```

PSJF

- The Preempted Shortest Job First policy runs the shortest of the currently available jobs. It allows for preemption, such that when a new, better job arrives, the scheduler will switch to new job
    - It should be logical that any “switch” will only occur when a new job arrive. So I only check for switching if a new job is spawned and not every unit time
- A priority queue is used in PSJF, the priority is set based on the execution time remaining (lowest first)
- The pseudocode for PSJF is as below:

```
PSJF{
    qsort() //sort by ready time
    init() //set the cpu, and proc states
    pq = createPriorityQueue()
    running_process = NULL
    while(1){
        if(running process is done running){
            terminate process
            if (all process done){
                exit(0)
            }
        }
        if(another process can be spawned){
            spawn a new process
            add to priority queue
            if(new process better than current process){
                run new process
                add current process into pq
            }
        }
        if(no process is running now){
            running_process = pq.pop()
        }
        if(process is selected){
            process = active
            process = max priority
        }
        unit time()
        process exec time --
    }
}

```

Final Notes

- Two if statements are added in preempted versions that deals with setting selected but not running process. This is to double check as preemption may mess up the ways in which a new system starts running

Comparison of Results

Notes

- I will attempt to compare the results for the 4 test cases as specified in DEMO.
- Since I don't really know how to plot using python or matlab, I will be using google sheets and tricks. More on the methodology later
- In the numerical comparison section, I am comparing the end time of the ideal and actual forked process.
    - I think end time better represent the actual duration it runs in case of weird CPU scheduling bursts (ex. one process gets a unit randomly before context switching out again)
- Error and Error rate is calculated by the formulas:
    - $Error = Idealendtime - realendtime$
    - $Error\% = (ideal - real)/ideal * 100$

Time Measurement

- Methodology: Run main.out on TIME_MEASUREMENT.txt, and average the 10 processes divide

by 500 to get how long an average tick is

```
P0 23372
P1 23373
P2 23374
P3 23375
P4 23376
P5 23377
P6 23378
P7 23379
P8 23380
P9 23381

[40241.669954] [Project1] 23372 1587911858.701167936 1587911859.597331010
[40243.510711] [Project1] 23373 1587911860.515233307 1587911861.438093260
[40245.345744] [Project1] 23374 1587911862.361411988 1587911863.273133035
[40247.167505] [Project1] 23375 1587911864.158625024 1587911865.094900224
[40249.020853] [Project1] 23376 1587911866.029510584 1587911866.948254231
[40250.875423] [Project1] 23377 1587911867.870158217 1587911868.802831148
[40252.711328] [Project1] 23378 1587911869.701709697 1587911870.638742370
[40254.622502] [Project1] 23379 1587911871.567760362 1587911872.549923606
[40256.435952] [Project1] 23380 1587911873.457732545 1587911874.363378609
[40258.297437] [Project1] 23381 1587911875.275094170 1587911876.224869904
```

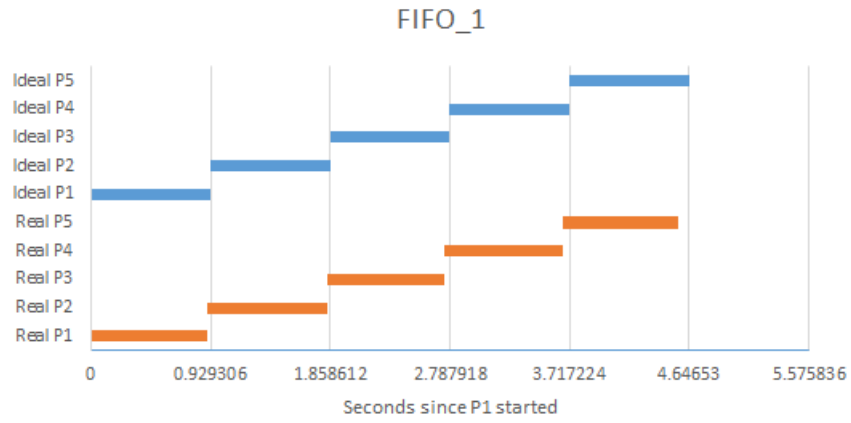
Average time per process = 0.929306006
Average unit time = 0.001858612

FIFO_1.txt

- Each process should take `average unit time` * `exec time` number of seconds.

```
Input:
FIFO
5
P1 0 500
P2 0 500
P3 0 500
P4 0 500
P5 0 500

Output:
P1 22834
P2 22835
P3 22836
P4 22837
P5 22838
[39572.736519] [Project1] 22834 1587911189.754355770 1587911190.660550871
[39573.667476] [Project1] 22835 1587911190.660721140 1587911191.591525322
[39574.577985] [Project1] 22836 1587911191.591664251 1587911192.502051782
[39575.496157] [Project1] 22837 1587911192.502182947 1587911193.420241107
[39576.395937] [Project1] 22838 1587911193.420375168 1587911194.320037096
```



- I've created an excel graph that shows approximately how my process runs compared to the ideal process deduced by the calculation earlier
- Some numerical Comparisons:

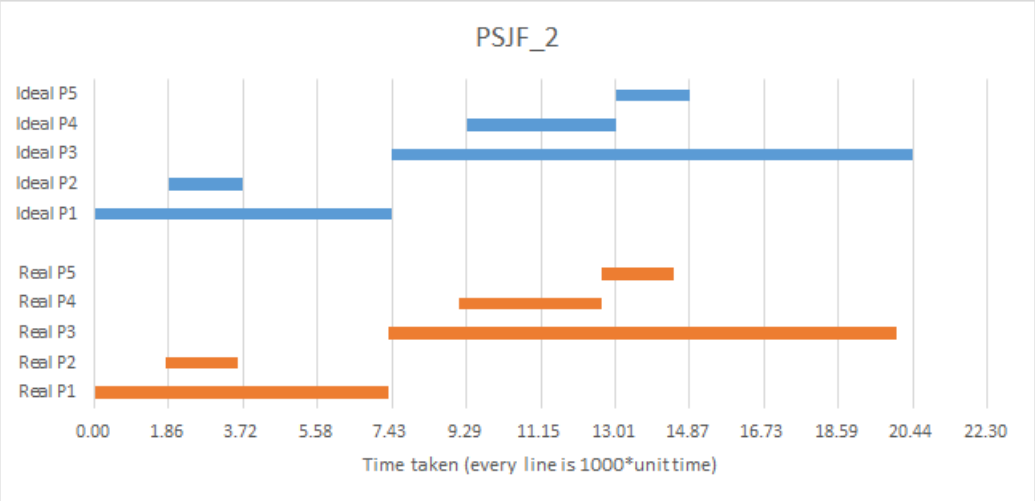
Process	Ideal	Real	Error	Error %
P1	0.929306	0.9062	0.023106	2.486354
P2	0.929306	0.9308	-0.001494	-0.16076
P3	0.929306	0.91039	0.018916	2.035509
P4	0.929306	0.91806	0.011246	1.21017
P5	0.929306	0.89966	0.029646	3.190137
		Average	0.016284	1.752282

PSJF_2.txt

- Hand calculated values that corresponds to ideal

```
Input:
PSJF
5
P1 0 3000
P2 1000 1000
P3 2000 4000
P4 5000 2000
P5 7000 1000

Output:
P1 23032
P2 23042
P3 23046
P4 23056
P5 23072
[39865.418725] [Project1] 23042 1587911481.510008448 1587911483.345029518
[39869.158666] [Project1] 23032 1587911479.748823005 1587911487.084978861
[39874.502007] [Project1] 23056 1587911488.841011346 1587911492.428332176
[39876.300480] [Project1] 23072 1587911492.428713052 1587911494.226808429
[39881.863324] [Project1] 23046 1587911487.085152350 1587911499.789665093
```



- I've created an excel graph that shows approximately how my process runs compared to the ideal process deduced by the calculation earlier
- Every tick is 1000 unit time
- Some numerical Comparisons:

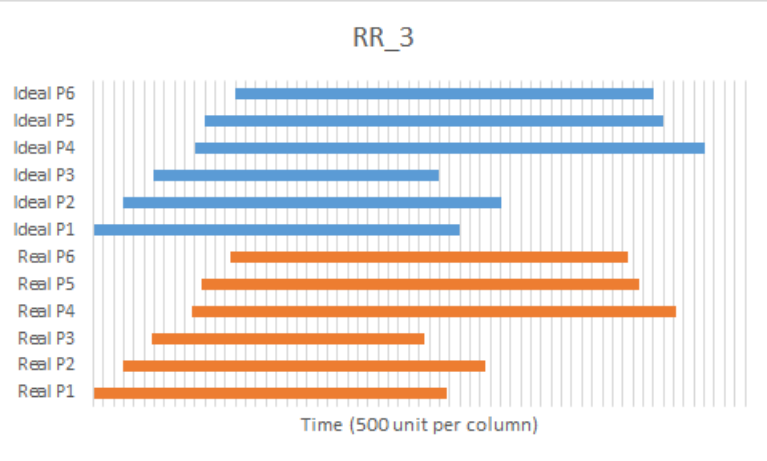
Process	Ideal	Real	Error	Error %
P1	7.434448	7.33615	0.098298	1.322198
P2	3.717224	3.5962	0.121024	3.255764
P3	20.44473	20.04084	0.403892	1.975532
P4	13.01028	12.67951	0.330774	2.542406
P5	14.8689	14.47798	0.390916	2.629087
		Average	0.268981	2.344997

RR_3.txt

- Since I don't know when my program "starts", I can't really count the first 1200 time unit where round robin doesn't schedule anything. I will therefore use 1200 as the starting point, and adjust all ready time by `ready_time-=1200`.
- Note that after this adjustment, ready time of p5 becomes 4000. My program will add it to queue before the current process is "recycled".
    - In other words, when P1 is done, and wants to be inserted after P4, P5 is first inserted before P1. The queue then looks like `P2->P3->P4->P5->P1`

```
Input:
RR
6
P1 1200 5000
P2 2400 4000
P3 3600 3000
P4 4800 7000
P5 5200 6000
P6 5800 5000

Output:
P1 26643
P2 26648
P3 26662
P4 26671
P5 26672
P6 26673
[59208.423055] [Project1] 26662 1587965147.864252931 1587965172.813845499
[59210.396962] [Project1] 26643 1587965142.530445047 1587965174.787749261
[59213.943330] [Project1] 26648 1587965145.250197199 1587965178.334113428
[59226.915400] [Project1] 26673 1587965155.116422453 1587965191.306156452
[59227.900640] [Project1] 26672 1587965152.443684083 1587965192.291404436
[59231.388307] [Project1] 26671 1587965151.525809306 1587965195.779066531
```



- The graph aboveshows clearly that the real and ideal process have a very similar overall trend/spread (as expected if my scheduler is correct), but the real process runs faster (magnitude of up to 1500 unit time)
  - Do note that the “end time” as represented by orange is the time the child process end, not the time the scheduler decide to wait for it (which would be basically ideal)
- Some numerical Comparison

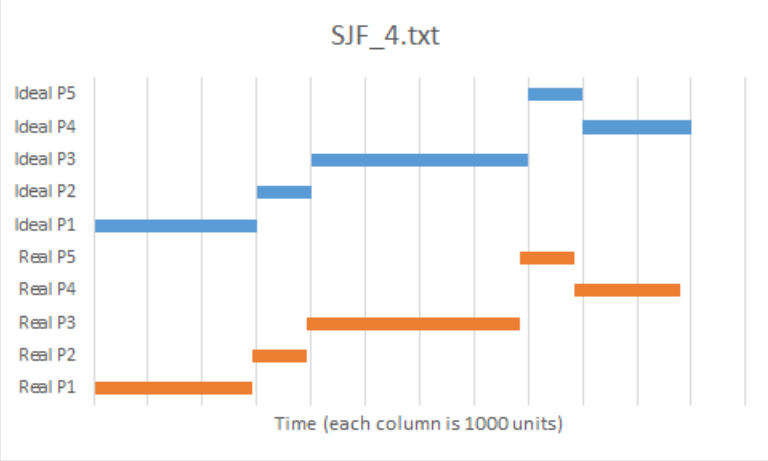
Process	Ideal	Real	Error	Error %
P1	33.45501623	32.2572999	1.197716331	3.580079959
P2	37.17224026	35.80366993	1.368570328	3.681699888
P3	31.59640422	30.28339982	1.313004398	4.155550072
P4	55.75836039	53.24861979	2.509740591	4.50110185
P5	52.04113636	49.76095986	2.280176497	4.381488676
P6	51.11183035	48.77570987	2.336120486	4.570606198
		Average	1.834221439	4.145087774

SJF_4.txt

- No preemption needed, each round select the shortest exec time process and run until finished.

```
Input:
SJF
5
P1 0 3000
P2 1000 1000
P3 2000 4000
P4 5000 2000
P5 7000 1000

Output:
P1 30719
P2 30729
P3 30730
P4 30731
P5 30732
[65344.520583] [Project1] 30719 1587971303.489549815 1587971308.933135457
[65346.347512] [Project1] 30729 1587971308.933365393 1587971310.760185489
[65353.656017] [Project1] 30730 1587971310.760346826 1587971318.069161073
[65355.527623] [Project1] 30732 1587971318.069586658 1587971319.940888870
[65359.121220] [Project1] 30731 1587971319.941089571 1587971323.534715198
```



- Numerical Analysis:

Process	Ideal	Real	Error	Error %
P1	5.575836039	5.439469814	0.136366224	2.445664172
P2	7.434448051	7.212879896	0.221568155	2.980290585
P3	14.8688961	14.38082981	0.488066292	3.282464875
P4	20.44473214	19.79210997	0.652622175	3.192128763
P5	16.72750812	16.18938994	0.538118172	3.216965539
		Average	0.407348204	3.023502787

Conclusion and Discussion

Things I did well

- Overall, the scheduling policies looks correct in terms of their ordering
- For the four tests I ran, their error rate are all relatively low

Things to improve

- I should be making a better, automated way to check and create graph. However, I don't have enough expertise, and so created all my graphs and tables in excel (check the report_excel file)
- In my debugging, I have seen a few cases where the high priority job finishes before the scheduler, and so the CPU automatically scheduled the next job (out of my control)
    - One way to control this behavious is by making the "next" process having a medium priority, thus whenever the real high priority process finishes, the scheduler will choose the correct next process to run

Analysis of error

- Forked process runs faster than scheduler
    - This is a result of each "time unit", my scheduler have to check more things, whereas the child just have to run a timeunit() function
    - Logically, my scheduler takes longer per unit time, and therefore the real process start and end time is faster
    - While there may be more overhead due to the forked process being switched around, overall it seems that scheduler still runs slower
- Synchronization of cores
    - The forked cpu and scheduler runs on two different CPU. These two CPU are not neccessarily synchronized - problems as small as a manufacturing difference may possibly affect the speeds of each CPU
    - The cores never communicate with each other, so there was no way that 500 unit time of one is the same as 500 unit time of another
- CPU allocation
    - Before I started running this program, there are already other processes running on all of the CPU. The processes are obviously not identical, so the load of each CPU may further affect the speed on each core