[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_OTHER, 0)
}

medPriority(pid){
    sched_setscheduler(pid, SCHED_OTHER, 0)
    nice(-20);
}


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

bufferProcess(){
    if (child){
        setcpu(FORK_CPU)
        medpriority
        infinite loop;
    }
    if (parent){
        setcpu(FORK_CPU)
        medpriority
    }
}

childProcess(proc){
    if (child){
        sched_yield();
        record start
        loop
        record end
        print
    }
    if (parent){
        setcpu(FORK_CPU)
        minPriority
        print name pid
    }
}
```

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{
    createBuff() //useful in many preemptions and stabilizing time
    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){
                kill buffer
                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{
    createBuff
    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){
                kill buff
                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

- A file named “error.txt” contains a numerical representation of everything. However, only below results were made into graphs
    - The error report generating program is written by Kaienlin, and I am simply using it to check my work
- 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
```

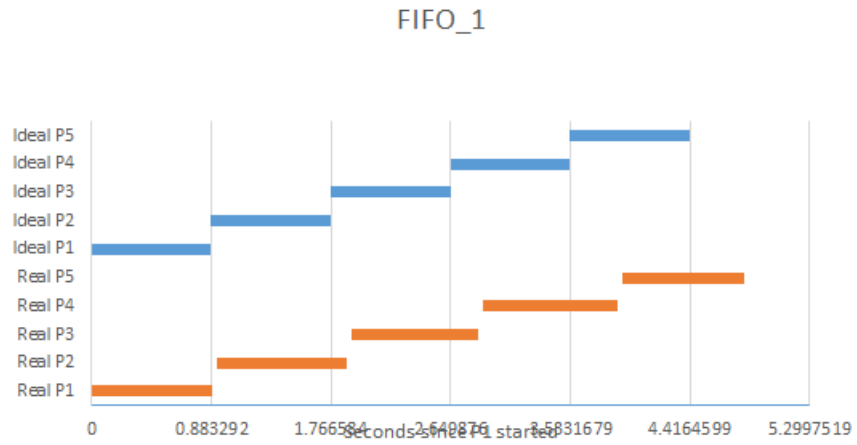
Average time per process = 0.929306006
Average unit time = 0.001858612

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 16366
P2 16367
P3 16368
P4 16369
P5 16370
[ 10420.985192] [Project1] 16366 1588062167.294876904 1588062168.182127997
[ 10421.986098] [Project1] 16367 1588062168.219540761 1588062169.183035571
[ 10422.955308] [Project1] 16368 1588062169.217554920 1588062170.152245508
[ 10423.982888] [Project1] 16369 1588062170.189252719 1588062171.179829336
[ 10424.920229] [Project1] 16370 1588062171.215347141 1588062172.117171723
```



- 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:
-

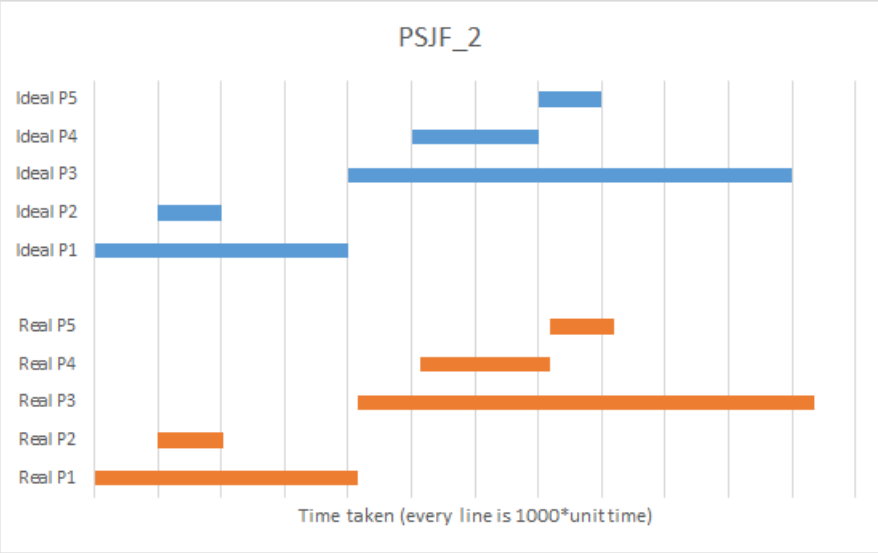
Process	Ideal	Real	Error	Error %
P1	0.929306	0.9062	0.023106	2.486354
P2	0.929306	0.9308	-0.001494	-0.16076
P3	0.929306	0.91039	0.018916	2.035509
P4	0.929306	0.91806	0.011246	1.21017
P5	0.929306	0.89966	0.029646	3.190137
		Total	0.08142	1.752282

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 14356
P2 14366
P3 14367
P4 14377
P5 14387
[ 9186.843564] [Project1] 14366 1588060932.291060468 1588060934.037701912
[ 9190.355908] [Project1] 14356 1588060930.502432294 1588060937.550066857
[ 9195.685099] [Project1] 14377 1588060939.328638025 1588060942.879290227
[ 9197.473106] [Project1] 14387 1588060942.921875272 1588060944.667307603
[ 9202.912052] [Project1] 14367 1588060937.583959347 1588060950.106286359
```



- 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:

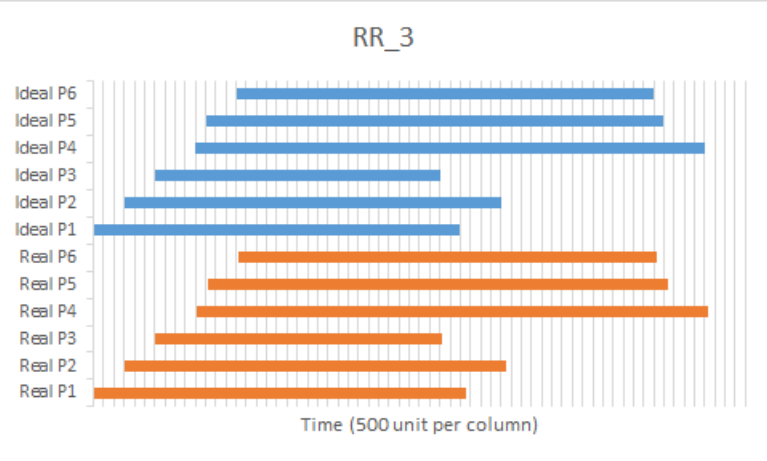
Process	Ideal	Real	Error	Error %
P1	7.066336	7.33615	-0.26981	-3.8183
P2	3.533168	3.5962	-0.06303	-1.78401
P3	19.43242	20.04084	-0.60842	-3.13093
P4	12.36609	12.67951	-0.31342	-2.53453
P5	14.13267	14.47798	-0.34531	-2.44333
		Average	-0.32	-2.74222

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

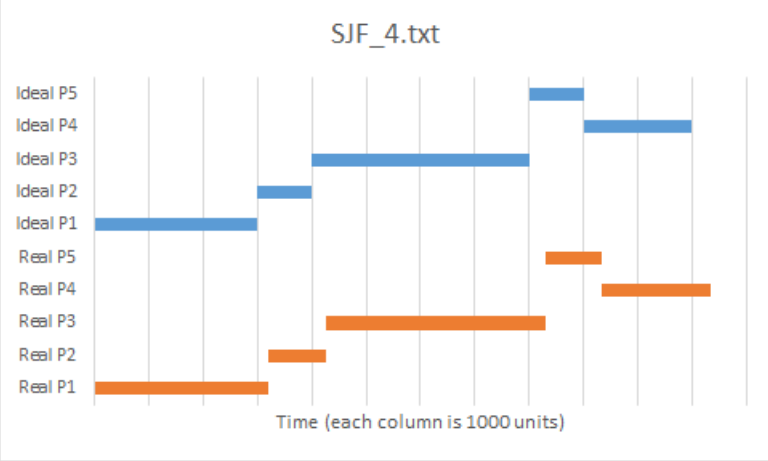
P1	**31.79851141**	**32.2572999**	**-0.45878849**	**-1.44279864**
P2	35.33167934	35.80366993	-0.471990585	-1.335884945
P3	30.03192744	30.28339982	-0.251472378	-0.837350111
P4	52.99751902	53.24861979	-0.251100779	-0.473797233
P5	49.46435108	49.76095986	-0.296608782	-0.59964151
P6	48.5810591	48.77570987	-0.194650769	-0.40067214
		Average	-0.320768631	-0.84835743

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 14706
P2 14716
P3 14719
P4 14729
P5 14739
[ 9536.084273] [Project1] 14706 1588061277.757753322 1588061283.279919110
[ 9537.868827] [Project1] 14716 1588061283.314579015 1588061285.064478201
[ 9544.909749] [Project1] 14719 1588061285.098123144 1588061292.105420598
[ 9546.704102] [Project1] 14739 1588061292.139841938 1588061293.899778447
[ 9550.241091] [Project1] 14729 1588061293.935813352 1588061297.436777523
```



- Numerical Analysis:

Process	Ideal	Real	Error	Error %
P1	5.299751902	5.628249884	-0.328497982	-6.198365284
P2	7.066335869	7.507229805	-0.440893936	-6.239357205
P3	14.13267174	14.68330979	-0.550638056	-3.896206365
P4	19.43242364	20.03574991	-0.603326273	-3.104740223
P5	15.8992557	16.46881986	-0.569564152	-3.582332169
		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