

# OS 2020 Project1

---

B06902074 資工三 柯宏穎

## Environment

---

OS: *Ubuntu16.04*

GCC version: 5.4.0

cmake version: 3.5.1(minimum required: 2.8)

python version: 3.5.2(for checking script, not necessary)

Virtual Hardware:

- 8GB RAM
- 6 Threads(*Host: i7-8750H, 6 Cores, 12 Threads*)

I use cmake generator to generate the *Makefile*, it might can't run on other machine. If you want to compile by `make`. Please run `cmake .` first to get the newest *Makefile*. How to run is described in *Readme*.

## Design

---

In this assignment, there are four part in my project:  
*main*, *process*, *scheduling* and *priority queue* respectively. I maintain a priority queue to make me scheduling more convenience. There are two kinds of priority, *process priority* and *CPU priority* respectively at the following description. One is my priority queue's priority, the other one is real CPU's priority(control by *sched\_setscheduler*).

My *process priority* is determined by its policy. If the policy is *RR* or *FIFO*, the priority is constant. Else, the priority is  $\frac{1}{\text{remain exec. time}}$  and it will update every *UNIT TIME* if it is running. Hence, even in preemptive mode, it will be switched because the priority is update frequently. *CPU priority* will be set to 99 if it is waked up, 1 if it is blocked.

1. Read the input and sort them by ready time.
2. *Parent(main)* start to check whether there is someone ready. If so, push it into priority queue.
3. Run the top of priority queue, context switch to other by the policy.
4. When context switching, if there is someone running, *block* it and *push* it into queue. Then *pop* the top and *wakeup* it.

5. If someone finish, *wait* it to avoid zombie exist too long.

There are some general information in my queue. Only one strange point is there is *remain\_time* and *available\_time* in it. The reason is *RR* need to switch every 500 *UNIT TIME*. *available\_time* is running remain time and *remain\_time* is real remain time. After *available\_time* reduce to 0, it need to change to other process. In other policy, this two will be the same.

If the priority is the same(*FIFO* and *RR*), it will perform as a general queue(first in first out). If there is a process stop and a process in at the same time, I will deal with stopping process. For example, In `RR_4.txt`, *P1* run until 500 and *P5*, *P6* arrive at 500, too. Then *P1* will push into the queue first, and *P5*, *P6* is the following.

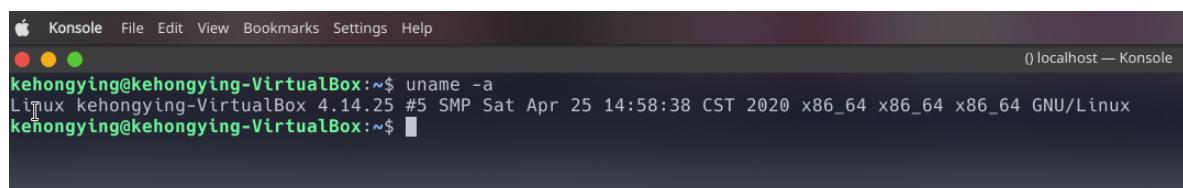
I use two cores to run *Parent* and *Child process* respectively. If we only use one, *Child process* might be preempted by *Parent process*. This will cause the running time incorrect. So we need two cores to run them respectively. In *Parent's* core, *main's CPU priority* is the highest to make sure it will be scheduled most frequently. When a process ready, It will be *fork()* and *block* immediately. When a process is *blocked*, it will be *assign* to *Parent's* core and reduce the *CPU priority* to 1, which is much smaller than *main*. If it is its turn to run, it will be *assign* to *Child's* core and run. At the begin, *Child process* is always in the *Child's* core, there is a problem: in some time period, all of process in *Child process* are blocked, which means they have the same *CPU priority*, some of them will be scheduled to run but it can't in fact. So in my method, all of blocked process will be placed in the *Parent's* core, and all of them will not be scheduled because their *CPU priority* is significant lower than *main*.

Even I use the above method to avoid child run in wrong time, I notice that some of them will sneak sometimes. This will cause they get the start time too early and the execution time we computed would be too long. So I set a new *while* loop to check whether its *CPU priority* has been rise or not. If not, it means that the process sneak and because parent hasn't let it run. It will be blocked in the loop until it wake up and rise the priority.

## Kernel Version

---

I use *Linux 4.14.25*, which is same to *HW1* and provided by the TA.



```
kehongying@kehongying-VirtualBox:~$ uname -a
Linux kehongying-VirtualBox 4.14.25 #5 SMP Sat Apr 25 14:58:38 CST 2020 x86_64 x86_64 x86_64 GNU/Linux
kehongying@kehongying-VirtualBox:~$ █
```

A screenshot of a Linux terminal window titled '(localhost — Konsole)'. The window shows the command 'uname -a' being run and its output. The output includes the kernel version '4.14.25', the build number '#5', the build type 'SMP', the date 'Sat Apr 25 14:58:38 CST 2020', the architecture 'x86\_64', and the distribution 'GNU/Linux'. The terminal window has a dark background with light-colored text and a standard Mac OS X style menu bar at the top.

# Experience

I try to compute my *UNIT TIME*'s real running time by `TIME_MEASUREMENT.txt` first.

```
[44776.059868] [Project1] 21208 1587748963.619857647  
1587748964.274602330  
[44777.349069] [Project1] 21209 1587748964.913345998  
1587748965.564012433  
[44778.645695] [Project1] 21210 1587748966.206974267  
1587748966.860845140  
[44779.937994] [Project1] 21211 1587748967.504002550  
1587748968.153351429  
[44781.237663] [Project1] 21212 1587748968.801994671  
1587748969.453229212  
[44782.538066] [Project1] 21213 1587748970.102086563  
1587748970.753843828  
[44783.832348] [Project1] 21214 1587748971.399933335  
1587748972.048330929  
[44785.127501] [Project1] 21215 1587748972.692772317  
1587748973.343692519  
[44786.428768] [Project1] 21216 1587748973.988960985  
1587748974.645167108  
[44787.726037] [Project1] 21217 1587748975.290399680  
1587748975.942646055
```

And I use a script to compute their real running time, which is in `check_file/` and usage is in `Readme`, we can get:

```
[0.          0.65474486] 0.6547448635101318  
[1.2934885 1.94415498] 0.6506664752960205  
[2.58711672 3.24098754] 0.6538708209991455  
[3.88414502 4.533494  ] 0.6493489742279053  
[5.18213701 5.83337164] 0.6512346267700195  
[6.48222899 7.13398623] 0.6517572402954102  
[7.78007579 8.42847347] 0.64839768409729  
[9.07291484 9.72383499] 0.6509201526641846  
[10.36910343 11.02530956] 0.6562061309814453  
[11.67054224 12.32278848] 0.6522462368011475
```

The average running time is about  $0.652s$  for 500 *UNIT TIME*. I'll assume it as my ideal time and compare with other test. My starting time is start at "real start to run" instead of "ready time". Hence, it is different to *around time*. There is some deviation can be caused by *CPU* is unstable in sometimes, other process(system process), and so on. Compare with my classmate, It seems that my deviation is

smaller. I think the reason is I give more cores to the virtual machine. There are more resource to compete. Then the cores I used has the lower probability to be preempted by real process.

I compute some test data's theoretical time and compare with real running time(normalize the start\_time to 0):

#### #FIFO\_3.txt

theoretical			experience		
start	finish	exec	start	finish	exec
P1 0	10.432	10.432	0	10.680	10.680
P2 10.432	16.952	6.520	10.713	17.365	6.652
P3 16.952	20.864	3.912	17.384	21.361	3.976
P4 20.864	22.168	1.304	21.376	22.703	1.327
P5 22.168	23.472	1.304	22.707	24.042	1.334
P6 23.472	24.776	1.304	24.044	25.377	1.333
P7 24.776	29.992	5.216	25.385	30.723	5.337

#### #RR\_4.txt

theoretical			experience		
start	finish	exec	start	finish	exec
P1 0	29.992	29.992	0	30.744	30.744
P2 0.652	26.080	25.428	0.659	26.834	26.175
P3 1.304	18.908	17.604	1.309	19.483	18.174
P4 1.956	7.172	5.216	1.958	7.519	5.562
P5 3.260	8.476	5.216	3.260	8.821	5.561
P6 3.912	9.128	5.216	3.909	9.472	5.563
P7 4.564	24.123	19.559	4.917	24.885	19.968

#### #PSJF\_1.txt

theoretical			experience		
start	finish	exec	start	finish	exec
P1 0	32.600	32.600	0	33.192	33.192
P2 1.304	20.864	19.560	1.314	21.197	19.883
P3 2.608	13.040	10.432	2.723	13.210	10.487
P4 3.912	7.824	3.912	4.057	7.990	3.933

We can notice that theoretical time usually faster than real time. The reason is in real world, there is lots of thing need to do. Besides the formal job, the process need to run lots of thing to check who ready, when to switch, waiting child process return, and so on. And this two policy will do lots of context switch, so the loading will be more heavy. Moreover, lots of system process will preempt the resource and these time will be counted in.

We can also find that *FIFO*'s has the smallest error because its loading is the lightest. It don't need to recompute the priority and don't need to context switch frequently. In contrast, *RR* and *PSJF* need to switch more frequently and need to do float compute.

By this compare, It seems that my implement is not bad. I've run a script to check my time difference to the ideal time. The average time error is about 2%, and the maximum error isn't exceed 10%.