[2024-Spring] Operating System Project #2 Instruction

Implementation and analysis of the virtual CPU and its scheduling

Advisor	Gyeongsik Yang (g_yang@korea.ac.kr) System Software Laboratory. **https://ss.korea.ac.kr**
Assistant	Hwiju Cho and Hyunho Lee (osta@os.korea.ac.kr)
Topic	Implement custom virtual CPU object in kernel and analyze process scheduling policies on it.
Due date(submission)	**2024. 5. 23. (Thursday) PM 11:59**
Recommended environment	Ubuntu 18.04.02 (64 bit), Linux kernel 4.20.11 (VM environment)
Contents	1. Purpose 2. Implementation Environment i. User Space ii. Kernel Space 3. User space: user-space program for requesting **ku_cpu** 4. **ku_cpu**: virtual CPU object 5. Experiment and analysis 6. Restrictions 7. Submission guidelines

1. Purpose

This project aims to create a new virtual CPU object in kernel space and to analyze the process scheduling policies applied to the virtual CPU object. We will implement a virtual CPU object that functions similarly to real-world CPUs. The virtual CPU object will be created and exist in kernel space. This object can be used (occupied) by user applications through a system call. From the perspective of the virtual CPU object, it should be initialized, wait for process requests for CPU usage, be allocated (used) by processes as requested under various scheduling policies, and maintain its state within the kernel.

2. Implementation environment

We will call the new virtual CPU object "**ku_cpu**." Note that in our system, we assume that our system contains only one **ku_cpu** (CPU core).

i) User space

A. Implement a user process that requests the use of a virtual CPU.
B. The processes use a system call for request virtual CPU requests.
C. Print response time and wait time

ii) Kernel space

A. Implement a virtual CPU object that behaves like a real CPU.
B. Can be implemented in a similar way to **Project #1** (through a system call handler).
C. Scheduling polices
- FCFS
- SRTF
- RR
- Priority with preemption

* See details for each below.

3. User space: user-space program for requesting ku_cpu

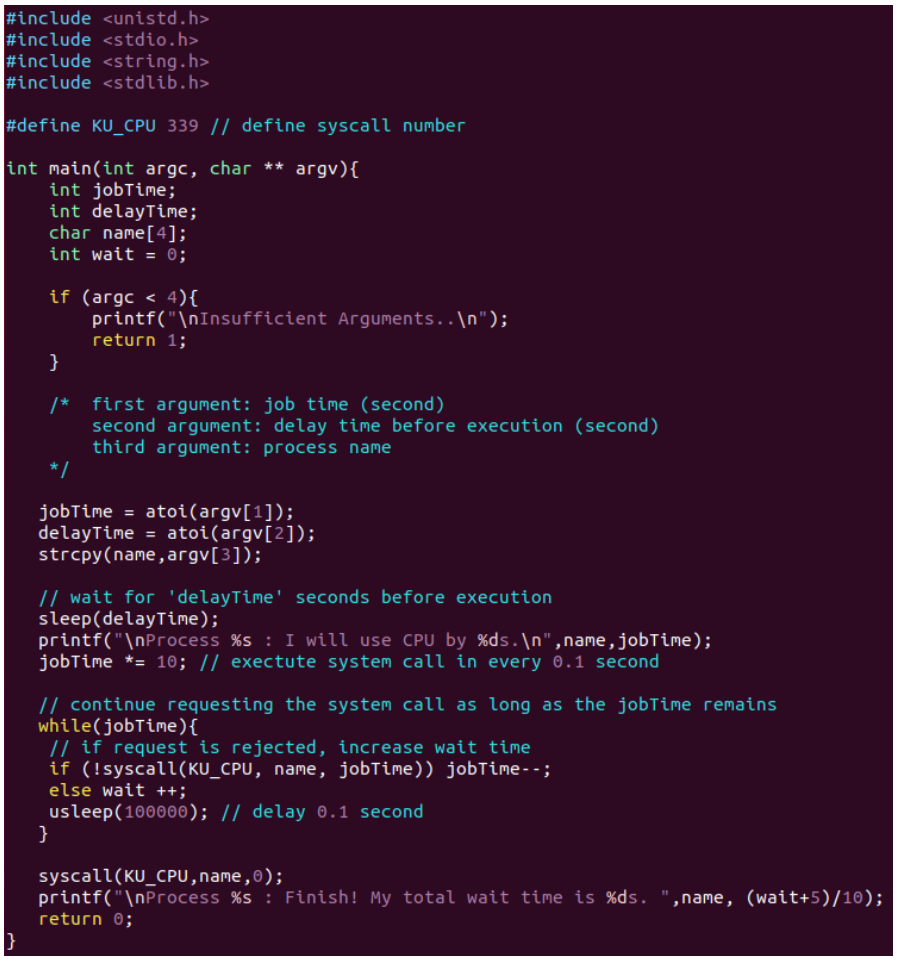
Our user program takes several arguments for its execution:

1. the execution duration (in seconds)
2. the starting delay (in seconds)
3. the name of the process
4. priority (If needed)

For example, we will run the program by giving the arguments "7 1 A" meaning that the process of name "A" will request the use of **ku_cpu** after 1 second from its execution and will use **ku_cpu** 7 seconds. And if we run the program with the arguments "7 1 A 1", it means that Its priority is 1.

We assume that when the request on the **ku_cpu** is succeeded by system call, 0.1 seconds of the execution duration is well handled. After using **ku_cpu** for the given execution duration, the process should print the total waiting time (with round-up).

Figure 1 shows the user space program implementation. Initially, **ku_cpu** system call number is defined with the **#define** statement. When calling the system call, the user program should pass 1) the process name (name) and 2) the remaining job time (job) as arguments and the 3) priority (if needed). The user process uses the return value to determine whether the request has been accepted or not. See the comments in the source code for more details.



```c
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define KU_CPU 339 // define syscall number

int main(int argc, char ** argv){
    int jobTime;
    int delayTime;
    char name[4];
    int wait = 0;

    if (argc < 4){
        printf("\nInsufficient Arguments..\n");
        return 1;
    }

    /*  first argument: job time (second)
        second argument: delay time before execution (second)
        third argument: process name
    */

   jobTime = atoi(argv[1]);
   delayTime = atoi(argv[2]);
   strcpy(name,argv[3]);

   // wait for 'delayTime' seconds before execution
   sleep(delayTime);
   printf("\nProcess %s : I will use CPU by %ds.\n",name,jobTime);
   jobTime *= 10; // exectute system call in every 0.1 second

   // continue requesting the system call as long as the jobTime remains
   while(jobTime){
    // if request is rejected, increase wait time
    if (!syscall(KU_CPU, name, jobTime)) jobTime--;
    else wait ++;
    usleep(100000); // delay 0.1 second
   }

   syscall(KU_CPU,name,0);
   printf("\nProcess %s : Finish! My total wait time is %ds. ",name, (wait+5)/10);
   return 0;
}
```

Figure 1: user-space program implementation without response time

Please note that the provided source code is incomplete. You should modify this code to **print both response time and wait time**. And you might need slight modification for priority scheduling.

4. ku_cpu: virtual CPU object

4.1 Operation sequence example

The following is the high-level operation sequence of **ku_cpu with FCFS:**

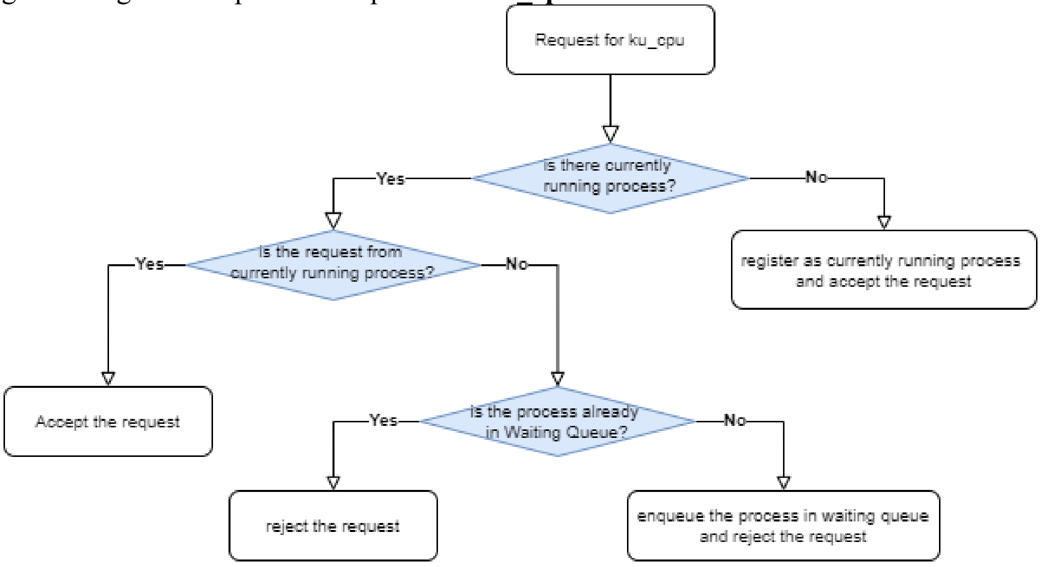

Request for ku_cpu
Is there currently running process?
Yes
No
register as currently running process and accept the request
Is the request from currently running process?
Yes
No
Accept the request
Is the process already in Waiting Queue?
Yes
No
reject the request
enqueue the process in waiting queue and reject the request

Figure 2: flow of **ku_cpu** usage.

1. The **ku_cpu** is implemented in kernel-space. User processes use a system call to request the use of the **ku_cpu** by providing two parameters: the process name, the time duration for occupying the **ku_cpu**. Upon this request, the **ku_cpu** in the system call handler executes the following tasks.
2. The **ku_cpu** keeps track of which process is currently running (occupying the **ku_cpu**):
   A. If the **ku_cpu** is idle (not occupied by any process), the first process that requests the **ku_cpu** is allowed to occupy it. In this case, the **ku_cpu** internally sets this process as the running process. Then, it returns 0 to notify the user process of successful **ku_cpu** occupation.
   B. If the process currently requesting the **ku_cpu** is the same one that is already occupying it, the system will return 0 to confirm the successful continuation of **ku_cpu** usage.
   C. If a different process requests the **ku_cpu** while it is occupied, the different process is added to the "waiting queue." However, if this process is already in the waiting queue, it will not be enqueued again. The system then returns 1 to notify the user process of the waiting status, indicating a temporary inability to occupy the **ku_cpu** immediately.
3. If the process currently using the **ku_cpu** has no more tasks remaining (the allocated duration has expired), it implies that the process has finished its execution on the **ku_cpu**. Subsequently, the next process is dequeued (popped) from the waiting queue and assigned the **ku_cpu**. If there is no process in the waiting queue, the **ku_cpu** reverts to an idle state.

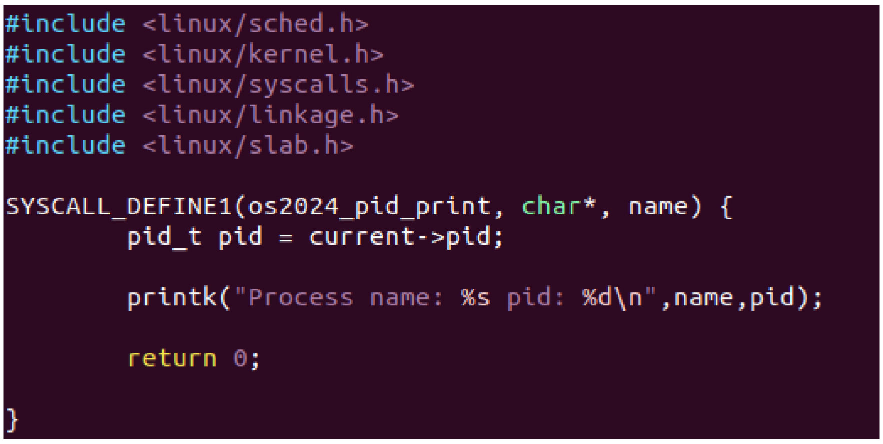
4.2 Data structure (or variables) for ku_cpu

To implement the **ku_cpu**, the following structures and variables are essential. You may use any type of implementation or data structures you required:

1. Waiting queue (processes waiting to use **ku_cpu**)
2. Variable to keep track of the current process occupying the **ku_cpu**
3. PID of the process being processed by **ku_cpu**

   To implement the **ku_cpu**, we should distinguish and identify the processes that called the system call. In Linux, a process is created through the **fork()** system call, and kernel assigns a unique PID value to each process. This PID is usually an integer value of 3 to 5 digits. The type of PID is **pid_t**, but it can be handled like an integer, and it can also be printed using **printk**.

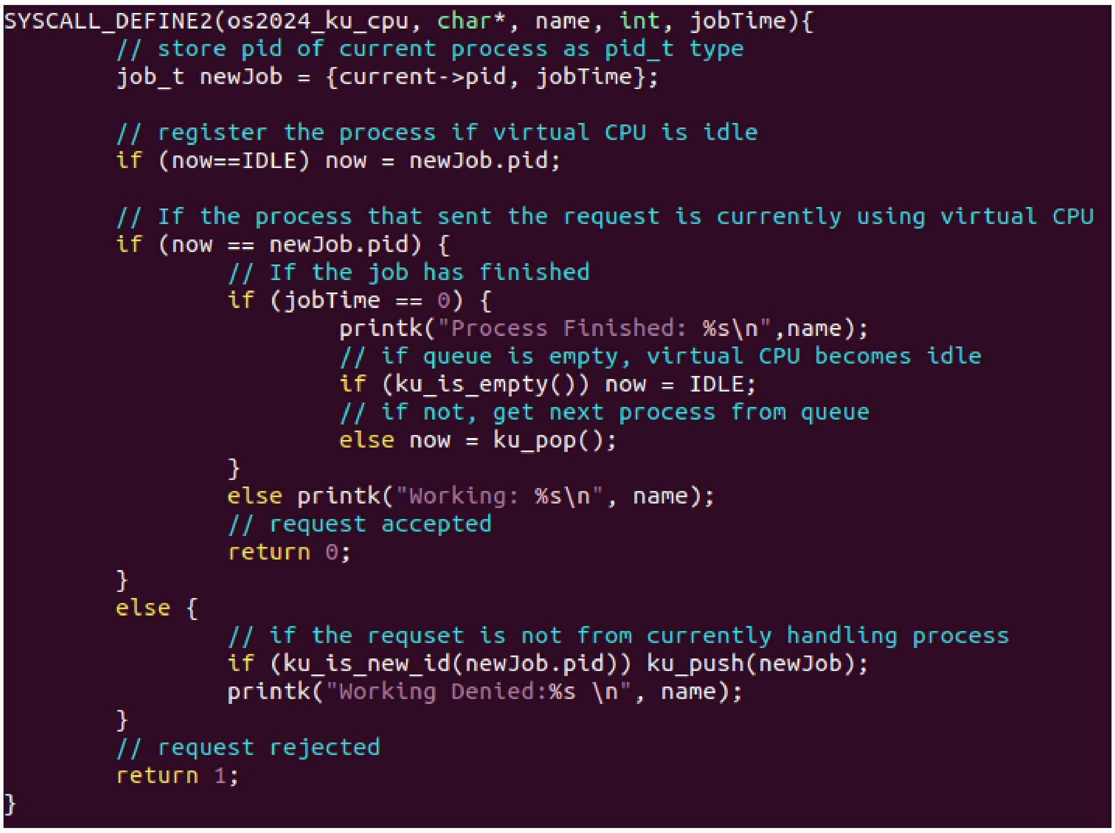
   We can obtain the PID from the PCB (process control block) structure in Linux, which is **task_struct**. Kernel provides a pointer to the **task_struct** structure of the current context. By adding the file **linux/sched.h** as a header file, we can get the PID value.



```
#include <linux/sched.h>
#include <linux/kernel.h>
#include <linux/syscalls.h>
#include <linux/linkage.h>
#include <linux/slab.h>

SYSCALL_DEFINE1(os2024_pid_print, char*, name) {
        pid_t pid = current->pid;

        printk("Process name: %s pid: %d\n",name,pid);

        return 0;

}
```

figure 3: print pid value

4.3 Example implementation of ku_cpu



```
SYSCALL_DEFINE2(os2024_ku_cpu, char*, name, int, jobTime){
        // store pid of current process as pid_t type
        job_t newJob = {current->pid, jobTime};

        // register the process if virtual CPU is idle
        if (now==IDLE) now = newJob.pid;

        // If the process that sent the request is currently using virtual CPU
        if (now == newJob.pid) {
                // If the job has finished
                if (jobTime == 0) {
                        printk("Process Finished: %s\n",name);
                        // if queue is empty, virtual CPU becomes idle
                        if (ku_is_empty()) now = IDLE;
                        // if not, get next process from queue
                        else now = ku_pop();
                }
                else printk("Working: %s\n", name);
                // request accepted
                return 0;
        }
        else {
                // if the requset is not from currently handling process
                if (ku_is_new_id(newJob.pid)) ku_push(newJob);
                printk("Working Denied:%s \n", name);
        }
        // request rejected
        return 1;
}
```

Figure 3: **ku_cpu** (by system call handler) implementation example.

Figure 3 is an example source code of a **ku_cpu** system call. Please note that the provided source code is incomplete and will not function if run as-is. It lacks several critical components, such as external or global variables and the implementation of the waiting queue. It is your responsibility to complete the remaining parts of the implementation:

4.4 Scheduling policies to implement

The above explanation is based on FCFS (first-come first-served) policy, known for its simplicity and ease of implementation. However, it has disadvantages as well. In this project, you are to start by fully implementing the FCFS scheduling policy. Once this is accomplished, you are expected to implement the additional scheduling policies:

1. First-come first-served (FCFS)
2. Shortest remaining time first (SRTF)
3. Round-Robin (RR)
4. Priority with preemption

* For detailed explanations and characteristics of each scheduling policy, please refer to our lecture notes.

5. Experiment and analysis

5.1 Experiment configuration

To validate our implementation, we will conduct experiments and report the results. For the experiment, run the user program as three separate processes concurrently and simultaneously. Assume that we have compiled the user program and obtained an executable file named **p**.

Then, create a new file named "**run**" with the following commands. You can adjust the execution duration or start delay to effectively reveal the differences between the scheduling policies.

- For FCFS, SRTF, RR

```
./p 7 0 A &
./p 5 1 B &
./p 3 2 C &
```

- For Priority

```
./p 7 0 A 2 &
./p 5 1 B 1 &
./p 3 2 C 3 &
```

The "&" symbol at the end of each command instructs the system to execute the process in the background in Linux. This allows us to execute all three **p** programs concurrently. You should try out these commands in the terminal to ensure you understand the procedure.

Next, enter the following command in the terminal to change the execution permissions of the **run** file. This command allows all users in Linux to read, write, and execute the **run** file:

```
$ chmod 777 run
```

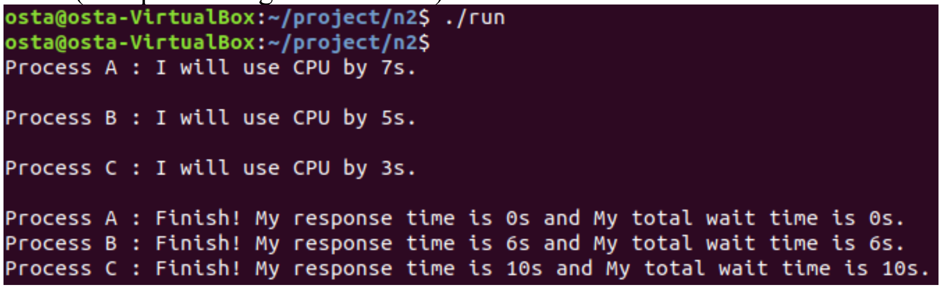
When executing the **run with FCFS**, process A will start immediately with a duration of 7 seconds. One second later, process B will start and run for 5 seconds. Following another second, process C will start and run for 3 seconds.

* You can modify this **run** file to conduct your own test for the report.

5.2 Expected results

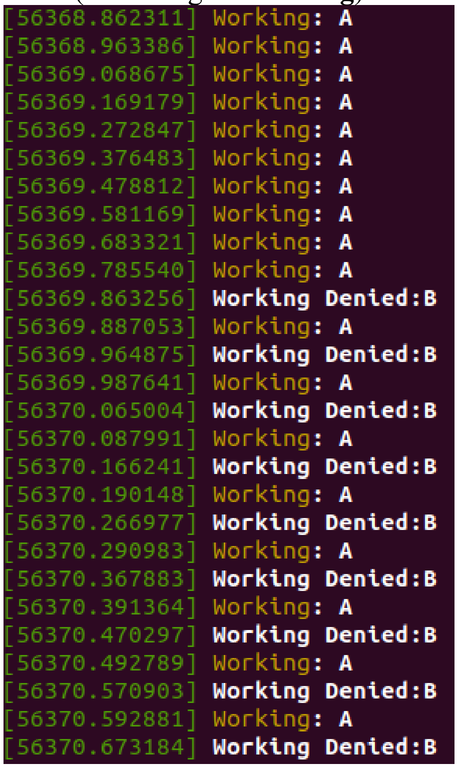
Note that your logs do not have to look exactly like the expected results below but should contain the necessary information on the scheduling, such as the wait time and the response time of each user process and working or working denied decisions from **ku_cpu**.

- FCFS (user process log from console)

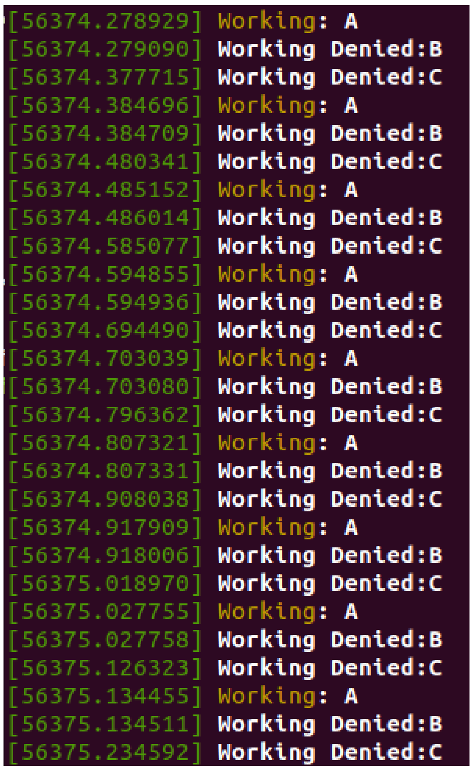


```
osta@osta-VirtualBox:~/project/n2$ ./run
osta@osta-VirtualBox:~/project/n2$
Process A : I will use CPU by 7s.

Process B : I will use CPU by 5s.

Process C : I will use CPU by 3s.

Process A : Finish! My response time is 0s and My total wait time is 0s.
Process B : Finish! My response time is 6s and My total wait time is 6s.
Process C : Finish! My response time is 10s and My total wait time is 10s.
```

- FCFS (kernel log from **dmesg**)

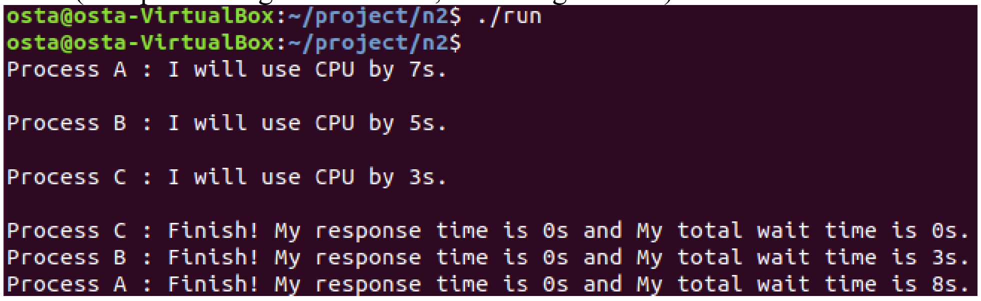


```
[56368.862311] Working: A
[56368.963386] Working: A
[56369.068675] Working: A
[56369.169179] Working: A
[56369.272847] Working: A
[56369.376483] Working: A
[56369.478812] Working: A
[56369.581169] Working: A
[56369.683321] Working: A
[56369.785540] Working: A
[56369.863256] Working Denied:B
[56369.887053] Working: A
[56369.964875] Working Denied:B
[56369.987641] Working: A
[56370.065004] Working Denied:B
[56370.087991] Working: A
[56370.166241] Working Denied:B
[56370.190148] Working: A
[56370.266977] Working Denied:B
[56370.290983] Working: A
[56370.367883] Working Denied:B
[56370.391364] Working: A
[56370.470297] Working Denied:B
[56370.492789] Working: A
[56370.570903] Working Denied:B
[56370.592881] Working: A
[56370.673184] Working Denied:B
```



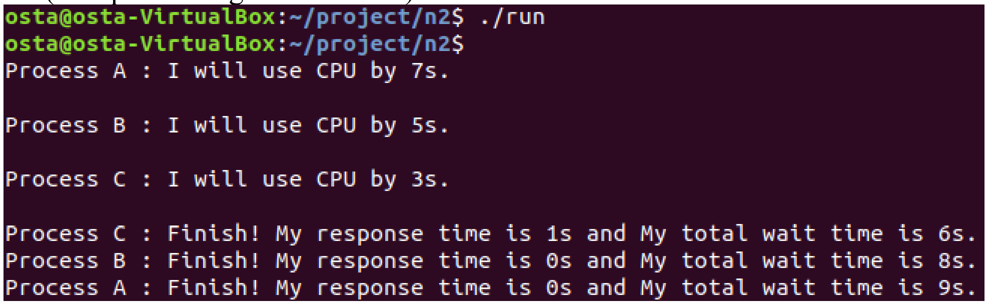
```
[56374.278929] Working: A
[56374.279090] Working Denied:B
[56374.377715] Working Denied:C
[56374.384696] Working: A
[56374.384709] Working Denied:B
[56374.480341] Working Denied:C
[56374.485152] Working: A
[56374.486014] Working Denied:B
[56374.585077] Working Denied:C
[56374.594855] Working: A
[56374.594936] Working Denied:B
[56374.694490] Working Denied:C
[56374.703039] Working: A
[56374.703080] Working Denied:B
[56374.796362] Working Denied:C
[56374.807321] Working: A
[56374.807331] Working Denied:B
[56374.908038] Working Denied:C
[56374.917909] Working: A
[56374.918006] Working Denied:B
[56375.018970] Working Denied:C
[56375.027755] Working: A
[56375.027758] Working Denied:B
[56375.126323] Working Denied:C
[56375.134455] Working: A
[56375.134511] Working Denied:B
[56375.234592] Working Denied:C
```

- SRTF (user process log from console, kernel log omitted)



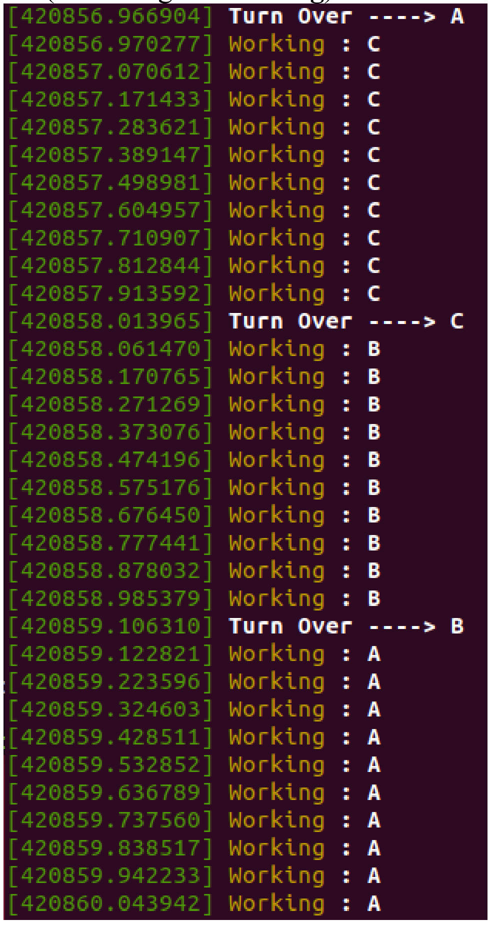
```
osta@osta-VirtualBox:~/project/n2$ ./run
osta@osta-VirtualBox:~/project/n2$
Process A : I will use CPU by 7s.

Process B : I will use CPU by 5s.

Process C : I will use CPU by 3s.

Process C : Finish! My response time is 0s and My total wait time is 0s.
Process B : Finish! My response time is 0s and My total wait time is 3s.
Process A : Finish! My response time is 0s and My total wait time is 8s.
```

- RR (user process log from console)

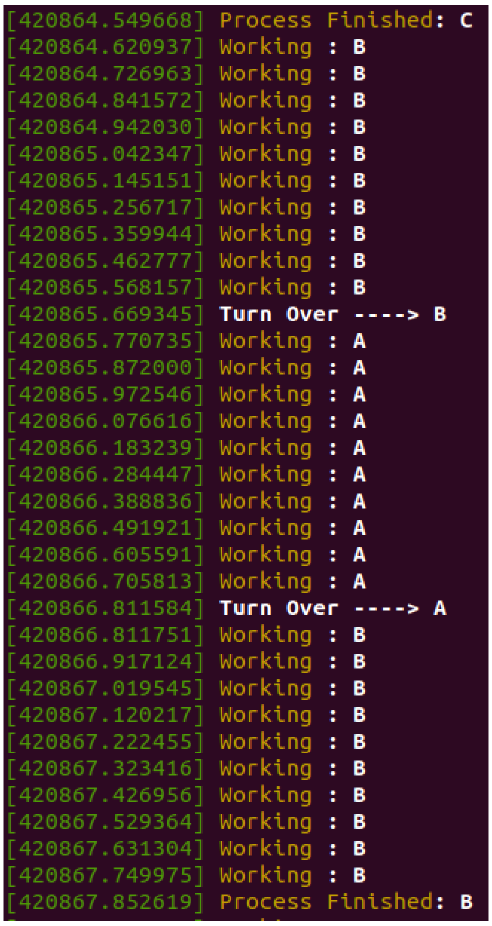


```
osta@osta-VirtualBox:~/project/n2$ ./run
osta@osta-VirtualBox:~/project/n2$
Process A : I will use CPU by 7s.

Process B : I will use CPU by 5s.

Process C : I will use CPU by 3s.

Process C : Finish! My response time is 1s and My total wait time is 6s.
Process B : Finish! My response time is 0s and My total wait time is 8s.
Process A : Finish! My response time is 0s and My total wait time is 9s.
```

- RR (kernel log from **dmesg**)

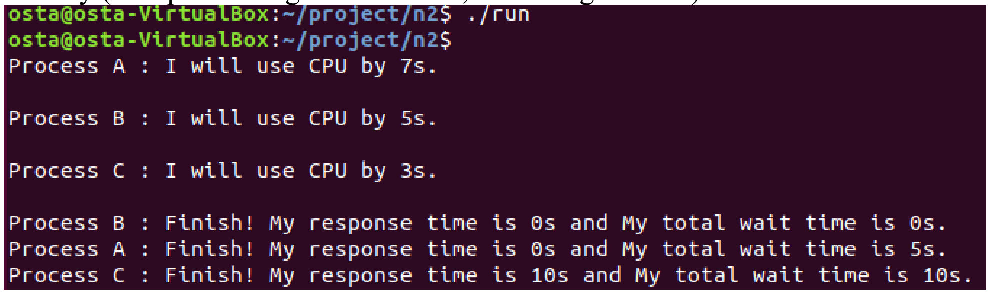


```
[420856.966904] Turn Over ----> A
[420856.970277] Working : C
[420857.070612] Working : C
[420857.171433] Working : C
[420857.283621] Working : C
[420857.389147] Working : C
[420857.498981] Working : C
[420857.604957] Working : C
[420857.710907] Working : C
[420857.812844] Working : C
[420857.913592] Working : C
[420858.013965] Turn Over ----> C
[420858.061470] Working : B
[420858.170765] Working : B
[420858.271269] Working : B
[420858.373076] Working : B
[420858.474196] Working : B
[420858.575176] Working : B
[420858.676450] Working : B
[420858.777441] Working : B
[420858.878032] Working : B
[420858.985379] Working : B
[420859.106310] Turn Over ----> B
[420859.122821] Working : A
[420859.223596] Working : A
[420859.324603] Working : A
[420859.428511] Working : A
[420859.532852] Working : A
[420859.636789] Working : A
[420859.737560] Working : A
[420859.838517] Working : A
[420859.942233] Working : A
[420860.043942] Working : A
```



```
[420864.549668] Process Finished: C
[420864.620937] Working : B
[420864.726963] Working : B
[420864.841572] Working : B
[420864.942030] Working : B
[420865.042347] Working : B
[420865.145151] Working : B
[420865.256717] Working : B
[420865.359944] Working : B
[420865.462777] Working : B
[420865.568157] Working : B
[420865.669345] Turn Over ----> B
[420865.770735] Working : A
[420865.872000] Working : A
[420865.972546] Working : A
[420866.076616] Working : A
[420866.183239] Working : A
[420866.284447] Working : A
[420866.388836] Working : A
[420866.491921] Working : A
[420866.605591] Working : A
[420866.705813] Working : A
[420866.811584] Turn Over ----> A
[420866.811751] Working : B
[420866.917124] Working : B
[420867.019545] Working : B
[420867.120217] Working : B
[420867.222455] Working : B
[420867.323416] Working : B
[420867.426956] Working : B
[420867.529364] Working : B
[420867.631304] Working : B
[420867.749975] Working : B
[420867.852619] Process Finished: B
```

- Priority (user process log from console, kernel log omitted)



```
osta@osta-VirtualBox:~/project/n2$ ./run
osta@osta-VirtualBox:~/project/n2$
Process A : I will use CPU by 7s.

Process B : I will use CPU by 5s.

Process C : I will use CPU by 3s.

Process B : Finish! My response time is 0s and My total wait time is 0s.
Process A : Finish! My response time is 0s and My total wait time is 5s.
Process C : Finish! My response time is 10s and My total wait time is 10s.
```

5.3 Analysis

In your analysis, you should compare the wait time and the response time of the processes under each of the scheduling policies. To effectively present your results, provide a graphical representation. Specifically, **draw a graph** that compares the wait times and the response time of the three processes across the different scheduling policies. Additionally, **discuss the characteristics** of each policy, such as their pros and cons, and the proper cases (situations) for the application of each policy.

6. Restrictions

1. In addition to the essential requirements, you can add any additional functions or error handling mechanisms if required. Please explain in the report why they are necessary)
2. Use the kernel log (**dmesg**) to watch the behavior of **ku_cpu**.
3. the queue in **ku_cpu** can be implemented as an array or any other structure type.
4. It is recommended to modify the source code examples in this instruction.
5. It is acceptable to have some discrepancies in the experiment results (including latency) as our project is a kind of simulation.

7. Submission guidelines

1. Submit all files through Blackboard.
2. List of files to submit
   A. Report
   B. Your source code: you should write comments for major (important) variables and functions
   C. Text file that contains execution results and log: save the log of user processes and scheduling-related kernel log
   D. **Video (e.g., mp4)** for executing "**run**" and observing logs: record the screen you are running the "run" process. Also, the video should include logs from the experiments. You can use a screen-capture function of your PC or any other convenient tool for the video recording.
3. How to submit files
   A. Gather source code files (and Makefile if any) into a directory named "source."
   B. Add the report, logs, and video files.
   C. Compress all the files and directories into "os2_[your student ID].tar" or "os2_[your student ID].tar.gz" or "os2_[your student ID].zip".

```
os2_[your student ID].zip
├── source
│   ├── source code 1
│   ├── source code 2
│   ├── source code 3
│   └── ...
├── report
├── log file 1
├── log file 2
├── log file 3
└── video
```

   D. **<u>Never submit the entire Linux kernel source code</u>.**
   E. All source codes must be able to be compiled and executed in an identical Linux environment.

4. Be sure to include the following sections in the report (The report should not exceed 10 pages, excluding the cover page):
   A. Please list the following on the cover: Department, Student ID (학번), Name, Submission date, Number of Freedays used for Project 2
   B. Development environment.
   C. Explanation of CPU scheduling and policies.
      i. **Caution:** Do not copy or paste existing material (including web).
      Although there is abundant data related to system calls, ensure you understand the content and write it in your own words.
   D. Implementation: modified and written codes with descriptions
      i. Do not attach the source code verbatim; instead, describe the overall workflow with important code pieces.
   E. Experiment results and your analysis with screenshots of execution results (logs).
   F. Problems encountered during the project and your solutions to them.

5. Note
   A. For late submissions that exceed the "Freeday" allowance (7 days in total), 1 point will be deducted from the total score for each day of late submission.
   B. **No project results will be accepted after two weeks (14 days) has passed from the deadline of the project #2.**