Operating System Project #2 Tutorial

Implementation and analysis of virtual CPU and its scheduling

2024. 5. 2.

Contents

Goal: Create a virtual CPU object and schedule processes

1

Project Introduction

What you need to implement

- **Implement virtual CPU and scheduling policies in kernel space**
  - Implement a virtual CPU object that behaves like a real CPU
  - Can be implemented in a similar way to Project #1 (through a system call handler)
  - Scheduling policies
    - FCFS (First-Come First Served)
    - SRTF (Shortest Remaining Time First)
    - RR (Round Robin)
    - Priority
- **Implement a simple user process using virtual CPU in user space**
  - Implement a user process that requests the use of a virtual CPU
  - The processes use a system call to request virtual CPU
  - Print response time and wait time (detail in Page 10)
- **See details in “Instruction.docx” at blackboard**

Virtual CPU: ku_cpu

- **The virtual CPU object (called ku_cpu*)**
  - It has variables for the scheduling operations
    - Waiting queue for scheduling
    - Variables to keep track of the current process occupying ku_cpu
    - pid of currently running process
    - ...
  - It gets the information of each process as arguments
- **A system call to use ku_cpu**
  - User process uses a system call to request ku_cpu
  - User process must pass their information (execution duration, starting delay, name, priority if needed) to ku_cpu syscall as arguments
  - Return value of system call: indicate whether the resource request has been accepted or not
    - return value 1: request rejected
    - return value 0: request accepted

* You can change the name if you want.

How ku_cpu performs as a system call handler

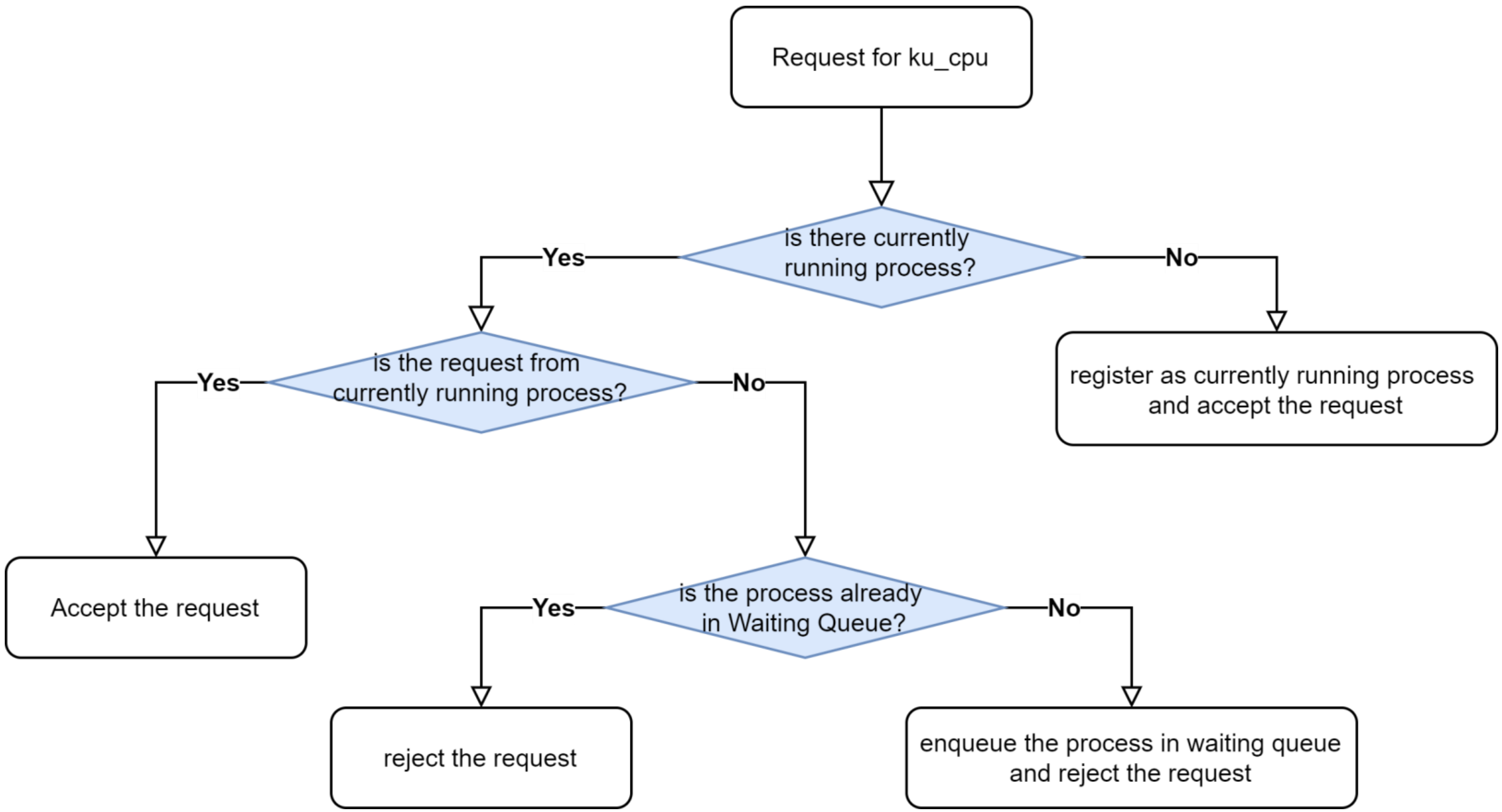
This example illustrates FCFS scheduling policy.

- **ku_cpu keeps track of which process is currently running on it (occupies the ku_cpu)**
- **If [ku_cpu == idle] (not occupied by any process),**
  - The first process that requests the ku_cpu is allowed to occupy it
  - ku_cpu internally sets this process as the running process
  - Return “0” to notify the user process of successful ku_cpu occupation
- **If [process currently requesting the ku_cpu**
  == process already occupying it],
  - Return “0” to confirm the successful continuation of ku_cpu usage

How ku_cpu performs as a system call handler

- **If [process currently requesting the ku_cpu**
  != process already occupying it],
  - The different process is added to the “waiting queue”
  - Note that if this process is already in the waiting queue, it will not be enqueued again
  - Return “1” to notify the user process of the waiting status, indicating a temporary inability to occupy the ku_cpu
- **If the current process of ku_cpu has no more tasks remaining:**
  - It means that the process has finished its execution on the ku_cpu
  - The next process is dequeued (popped) from the waiting queue and assigned to the ku_cpu
  - If there is no process in the waiting queue, the ku_cpu reverts to an idle state.

Flowchart of virtual CPU



This flowchart describes FCFS

Example source code: kernel space

- **Partial (skeleton) code**
  - It's incomplete source code
  - You must complete it by yourself

```
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;
}
```

Example source code: user space

- **User space code that uses ku_cpu**

– It's incomplete source code

– You must modify this code to

  - print both response time and wait time

  - support priority scheduling

```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;
}
```

Scheduling polices you should implement

- FCFS (First-Come First Served)
- SRTF (Shortest Remaining Time First)
- RR (Round Robin)
- Priority

Experiment configuration

- **Execute several user processes concurrently**
  - Assume you compiled the user program and created an executable named **"p"**
  - Then create a new file named **"run"** with the following commands
    - This type of file is called "script"

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

You can adjust **the execution duration or start delay** to effectively reveal the differences between the scheduling policies

    - The "&" after the command: a keyword to run the process in background
    - Using "&" allows us to execute three **`p`** concurrently

  - Enter the following command in the terminal

```
$ chmod 777 run
```

    - Changes the execution permissions of the file **run**
    - From now on, all users can read, write, and execute the file run.

Experiment configuration

- **Now, execute "`run`"**
  - When running the previous example commands for `run`,
    - Process A immediately runs with an execution time of 7 seconds
    - One second later, process B runs with a run time of 5 seconds
    - One second later, process C runs with a run time of 3 seconds

```
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.
```

(result with modified userspace code)

Example output (FCFS)

- **User space output**

```
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.
```

Example output (FCFS)

- **Kernel space output (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
```

Example output (FCFS)

- **Note that your logs do not have to be same as the example**
  - BUT should contain the necessary information about the scheduling (e.g., **wait time** of each user process and **working or working denied decisions** from ku_cpu)

```
Working: A
Working Denied:B
Working Denied:C
Working: A
Working Denied:B            repeated
Working Denied:C
Working: A
Working Denied:B
Working Denied:C
Process Finished: A
Working: B
Working Denied:C
Working: B
Working Denied:C            repeated
Working: B
Working Denied:C
Process Finished: B
Working: C
Working: C                  repeated
Working: C
Process Finished: C
```

```
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 : Finished! My total wait time is 0s.

Process B : Finished! My total wait time is 6s.

Process C : Finished! My total wait time is 10s.
```

2
Submission Policy

Submission Policy

- **Due date(submission): 2024. 5. 23 (Thursday) PM 11:59**
- **Submit the followings through blackboard**
  - Report
    - Details will be provided on a later slide
  - Your source code
    - Should write comments for major(important) variables and functions
    - Never submit the entire Linux kernel source code
  - Log text files
    - Save the log of user processes and scheduling-related kernel log
  - Video(e.g., mp4)
    - Record the screen you are running the “run” process
    - 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

Submission Policy

- **How to Submit**
  - Gather source code files (and Makefile if any) into a directory named “source”
  - Add the report, logs and video files
  - 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”.
- **Structure of submission file**

```
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
```

Report

- **Report should not exceed 10 pages (excluding the cover page)**
- **Report should contain the following contents**
  - Cover page with department, Student ID(학번), Name, Submission date, Number of Freedays used for Project 2
  - Development environment
  - Explanation of CPU scheduling and policies
    - Do not copy & paste existing material (including web)
  - Implementation: modified and written codes with descriptions
    - Do not attach the source code verbatim
    - Describe the overall workflow with important code pieces
  - Experiment results(log screenshot) and your analysis
  - Problems encountered during the project and your solutions to them

Others

- **Freeday: Total 7 days are given for projects #1 and #2**
  - For late submissions that exceed the "Freeday" allowance,
    1 point will be deducted from the total score for each day of late submission
  - No projects will be accepted **after two weeks (14 days)** passed from the deadline

3

Tips & FAQs

Current pointer & pid_t data type

- **pid_t**
  - A unique value that can distinguish processes in Linux
  - Integer value of 3 to 5 digits.
  - pid_t is data type for pid but can be handled as an integer
  - It can be printed by printk (just like integer value)
  - Can obtain the PID from the PCB (process control block) structure
    - The pid can be obtained through the pid member of the task_struct structure

- **current**
  - Pointer to task_struct of the current syscall requestd process to store which process is currently occupying ku_cpu
  - Should add linux/sched.h as a header file

Current pointer & pid_t data type

- **Sample code**

Kernelspace code

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

#define os2024_pid_print 338

int main(int argc, char **argv) {
        char name[4];

        if (argc < 2) {
                printf("\nInsufficient Argument..\n");
                return 1;
        }

        strcpy(name, argv[1]);

        syscall(os2024_pid_print, name);

        return 0;
}
```

Userspace code

```c
#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;

}
```

compile

```
osta@osta-VirtualBox:~/pidprint$ gcc pidprint.c -o pidprint
```

Current pointer & pid_t data type

- **Sample code - run**

```
osta@osta-VirtualBox:~/pidprint$ ./pidprint A
osta@osta-VirtualBox:~/pidprint$ ./pidprint A
osta@osta-VirtualBox:~/pidprint$ ./pidprint B
osta@osta-VirtualBox:~/pidprint$ ./pidprint B
osta@osta-VirtualBox:~/pidprint$ ./pidprint C
osta@osta-VirtualBox:~/pidprint$ ./pidprint C
```

```
[  102.372469] Process name: A pid: 2259
[  107.058611] Process name: A pid: 2260
[  109.200574] Process name: B pid: 2261
[  110.819210] Process name: B pid: 2262
[  113.367870] Process name: C pid: 2263
[  115.242981] Process name: C pid: 2264
```

$ dmesg

Scheduling polices

- **For RR (Round Robin)**
  - You can decide the value of time slice by yourself. Please mention the value you set in the report.
- **For Priority**
  - With the preemption
  - The smallest value has the highest priority

User space code for priority

- You can implement user space code for priority scheduling separately from other scheduling policies.

Exception handling

- **You can add any additional functions or error handling mechanisms *<u>if required</u>***
  - Please explain in the report why they are necessary.
- **<u>But this is not mandatory</u>**