2020 OS Project 1 — Process Scheduling

Goal

- Learn the concept of priority-driven scheduling and how to modify the Linux kernel.

Due Time

- 2020/04/29 23:59:59

Problem

Process scheduling is an essential part of an operating system supporting multiprogramming. There are various scheduling algorithms for different situations. However, many of them are based on the same concept: priority-driven scheduling. For priority-driven scheduling, it is important to assign an appropriate priority level to each process. The assignment depends on the desired scheduling policy. For example, a round-robin scheduler can be implemented by assigning the same priority level to all the processes. In this project, we would like to study the relation between priority assignment and scheduling policies.

Project

You are asked to design a user-space scheduler based on the priority-driven scheduler built in Linux kernel for a set of child processes. You may change priority of each child process directly for the specified scheduling policy by **sched_setscheduler**. The goal is to get a result as close to the expected schedule as possible. The more correct finish time and order of processes, the more points you will get. For more accuracy, you need to measure the start and finish time of each child process in Linux kernel. You might need to write your own system call based on the function **getnstimeofday** and show the information through **printk**.

In the main process, it first reads the input parameters from the standard input. The parameters will specify the scheduling policy, the number of child processes to create, and the set of child processes. The scheduling policy may be first in, first out (FIFO), round-robin (RR), shortest job first (SJF), or preemptive shortest job first (PSJF). In the input, each child process is described by a 3-tuple specifying its name, ready time, and execution time.

A process with a ready time **X** should be created via the system call **fork** after **X** units of time since the start of the main process. Similarly, a process with an execution time **Y** should execute, **not sleep**, **Y** units of time. The unit of time is defined as the execution time of an empty for-loop of one million iterations as follows.

```
{ volatile unsigned long i; for(i=0;i<1000000UL;i++); }
```

For convenience of debugging, each round (time quantum) of RR scheduler is defined as 500 time units in this project, and ready time of processes will not be the multiple of 500 time units. If the process finishes before the time quantum timer expires, then it is swapped out of the CPU just like FIFO algorithm.

Note that please use the following test set as the input when writing the report. However, TA will use other test cases during the demo.

- Donwload Test Set (https://www.dropbox.com/s/0sff5sr6mpqc3hn/OS_PJ1_Test.tar.gz?dl=0)

Prgram Input

- The program will get the input parameters from the standard input.

```
S // the scheduling policy, one of the following strings: FIFO, RR, SJF, PSJF.
N // the number of processes
N1 R1 T1
N2 R2 T2
…
Nn Rn Tn
//Ni - a character string with a length less than 32 bytes, specifying the name of the
//Ri - a non-negative integer, specifying the ready time of the i-th process.
//Ti - a non-negative integer, specifying the execution time of the i-th process.
```

For example, a input for FIFO scheduling with three processes might look like this:

```
FIFO
3
P1 1 10
P2 2 5
P3 2 7
```

Program output

- In each child process, show its name and PID to the standard output with the following format:

```
NAME PID
//NAME - the name of this process specified by the input
//PID - the process id of this process
```

- In Linux kernel, show the start and finish of each process by printk with the following format:

```
TAG PID ST FT
//TAG - the tag to identify the message of this project.
//PID - the process ID of this process
//ST - the start time of this process in the format seconds.nanoseconds.
//FT - the finish time of this process in the format seconds.nanoseconds.
```

The output of the program might look like this:

```
P1 4007
P2 4008
P3 4009
```

The output from the command **dmesg** might contain:

```
[Project1] 4007 1394530429.657899070 1394530430.004979285
[Project1] 4008 1394530429.668132665 1394530430.218223930
[Project1] 4009 1394530429.676547365 1394530430.221544405
```

Technical Issue

Usually, your system might have more than one cores. In this case, the underlying Linux kernel scheduler might not assign your tasks to the cores you expect. In order to solve the above problem, you can simply set your virtual machine to be single core or use sched_setaffinity to assign a task to a specific core.

Tasks set by **sched_setscheduler** will have higher priority than others, so you can focus on optimizing accuracy.

Time Measurement

The following test data will be used to determine one unit of time in your machine.

```
FIFO
10
P0 0 500
P1 1000 500
P2 2000 500
P3 3000 500
P4 4000 500
P5 5000 500
P6 6000 500
P7 7000 500
P8 8000 500
P9 9000 500
```

We will measure the time unit for this project in your machine by the average execution time of these 10 processes divided by 500. The performance of your scheduler will be determined by the difference between your output converted (by TA) in the time unit and the theoretical result.

Environment

Platform	OS	Programming Language	Machine
A real or virtualized PC or NB.	Linux	C	try to find one

Grading

- For each scheduler policy, your performance will be judged by the following 2 parts: The order (70%) of finish time and the converted finish time (30%). How close to reality!
- **Late penalty** is 10% per day. E.g. if 2 days late and with grade 90, the final grade will be 90 * 90% * 90% = 72.9.

Submit

- 請開啓一個 github repository 來管理本份作業，並至 表單 (https://forms.gle/Q3TKBgc269ARmvCC7) 中填寫 repository 的網址，助教將會下載該 repository 的內容進行評分。同時，遲交的天數也會以 repository 最晚 commit 的時間戳記來計算。

- repository 中需有一份報告（命名爲 report.pdf），報告請包含
  –1. 設計
  –2. 核心版本
  –3. 比較實際結果與理論結果，並解釋造成差異的原因

- 原始碼（Makefile, *.c, *.h）也請置於 repository

- 需與核心一同編譯的程式碼，請額外開一個名爲 kernel_files 的目錄來存放,另外也請在此資料夾內放上你的核心system call table及syscall header file,通常你會需要修改這兩個檔案來新增你的system call.

- 創建一個output資料夾，依照**所有**test set(總共21個)測試檔檔名來生成兩個檔案分別對應該次執行結果的standard output及dmesg內容。
  舉例:
  輸入FIFO_1.txt 我們必須產生FIFO_1_stdout.txt 及 FIFO_1_dmesg.txt
  你可以透過:

  ```
  ./yourProgram < FIFO_1.txt > FIFO_1_stdout.txt
  dmesg | grep Project1 > FIFO_1_dmesg.txt
  ```

  之類的方法生成檔案。
  為了讓TA可以改的輕鬆點，檔名的命名方法還請務必依照上述範例來命名。

- 由於疫情的關係無法現場DEMO，所以必須上傳一份五分鐘內的影片到demo的資料夾，影片內容僅需包含:
  –1.TIME_MEASUREMENT.txt
  –2.FIFO_1.txt
  –3.PSJF_2.txt
  –4.RR_3.txt
  –5.SJF_4.txt
  執行以上五個測試檔的影片(從下指令到程式結束，並且在每次執行完後
  `dmesg |grep Project1`)
  可以的話影片盡量在每次 `dmesg` 前先 `sudo dmesg clean`

- 最後，你的 repository 的目錄結構應該類似於：

```
repository
├── kernel_files
│   └── ...
├── XX.c
├── XX.h
├── Makefile
├── report.pdf
├── output
├── demo
└── ...
```

TA

當我們改完你的作業我們會寄一封信到你的ntu mail提醒你，
若你有關於Project1的相關問題，還請寄信到 osproject3310@gmail.com (mailto:osproject3310@gmail.com) or lab408TA@gmail.com (mailto:lab408TA@gmail.com) 告知我們
格式如下:

```
Subject: Project1 學號
---
姓名:小智
學號:b0xxxxxxx
問題:AAAAAAAAAAAAA
```

Hint

- The system call ***nice*** ***(http://linux.die.net/man/2/nice)*** .
- The system call ***schedsetaffinity*** ***(http://linux.die.net/man/2/sched_setaffinity)*** .
- The system call ***schedsetscheduler*** ***(http://linux.die.net/man/2/sched_setscheduler)*** .
- The command ***renice*** ***(http://linux.die.net/man/8/renice)*** .

Question

1. converted finish time (30%)怎麼計算的?

   我們會根據Time Measurement (如果你的程式看起來有疑慮我們會自己編並跑一遍)算出unit time，並將所有由你生成的output檔案依據unit time算出跟理想排序時間的誤差(一樣有問題我們會跑其他測試集)，最後根據大家的誤差時間來給分，給分方式會看你的時間誤

差有沒有超過我設的臨界值，如果比臨界值表現的好就會有基本分15%，之後根據你的排名表現給予分數(另外15%)。

2. start time紀錄甚麼?

   紀錄你的子行程被第一次丟上cpu執行時的時間，
   由於之前定義的很模糊，如果你已經實作了用行程的產生時間當成start time也不用改，我們會根據你用哪個時間當start time，來跑跟完美答案的誤差，不會影響到你的分數。

3. The order (70%) of finish time
   其中包含20%給報告(根據要求的完成度)， 40%測試結果順序有無錯誤(依比例扣) ，10%程式完成度

4. 可否使用兩顆cpu,例如:一顆排程，一顆專門跑子行程?
   可以，也建議

5. dmesg的timestamp可以留著嗎?
   可以