Project TA - Peter (cspeter@cse.ust.hk): Handling questions before the deadline
Grader TA - Kaiqiang (kxuar@cse.ust.hk): Grading and appeal handling after the deadline

COMP3511 Operating System (Spring 2022)

PA2: Simplified Linux Completely Fair Scheduler (CFS)

Released on 21-March-2022 (Monday) Due on 10-April-2022 (Sunday) at 23:59

Introduction

The Linux kernel implements a completely fair scheduler (CFS). Upon completion of the project, students should be able to understand the details of CFS (a **process scheduling** algorithm used in Linux operating system) and how to implement a simplified CFS.

Please note that the details of CFS are not covered in the lecture notes. This project description describes the detailed requirements of implementing a simplified CFS. In addition, you are highly recommended to attend the corresponding project-related lab.

Program Usage

You need to implement a program that simulates a CFS to schedule several processes. The program name is `cfs`. Here is the sample usage:

```
$> ./cfs < input.txt > output.txt
```

`$>` represents the shell prompt.
< means input redirection, which is used to redirect the file content as the standard input
> means output redirection, which is used to redirect the standard output to a text file
Thus, you can easily use the given test cases to test your program and use the `diff` command to compare your output files with the sample output files.

Getting Started

`cfs_skeleton.c` is provided. You should rename the file as `cfs.c`

You can add new constants, variables, and helper functions
Necessary header files are included. You should not add extra header files.

Assumptions

- There are at most 10 different processes
- There are at most 300 steps in the Gantt chart

Some constants and helper functions are provided in the starter code. Please read the skeleton code carefully.

Input Format

The input parsing is given in the skeleton code. It is useful to understand the input format.

All values are either integers or strings

You can assume that all values are valid
- For example, `num_process` must be a positive integer less than or equal to 10, and so on…

Empty lines and lines starting with # are ignored
- Without these comment lines, it is very hard to understand the input file

Format of constant:
- `name = <value>`

Format of vector (i.e., an array of value):
- `name = <values of the vector>`

A sample input file:

```
# COMP3511 PA2 (Spring 2022)
# An input file for a Simplified Completely Fair Scheduler (CFS)
# Empty lines and lines starting with '#' are ignored

# assume we have 2 processes
num_process = 2
sched_latency = 48
min_granularity = 6

# Example:
# P0: burst time is 60, nice value is -5
# P1: burst time is 30, nice value is 0 (default)

burst_time = 60 30
nice_value = -5  0
```

Output Format

The output consists of 3 regions:

- Displaying the parsed values from the input
- Displaying the intermediate steps of the CFS algorithm
- Displaying the final Gantt chart

The sample output file based on the sample input file:

```
=== CFS input values ===
num_process = 2
sched_latency = 48
min_granularity = 6
burst_time = [60,30]
nice_value = [-5,0]
=== CFS algorithm ===
=== Step 0 ===
Process Weight  Remain  Slice   vruntime
P0      3121    60      36      0.00
P1      1024    30      11      0.00
=== Step 1 ===
Process Weight  Remain  Slice   vruntime
P0      3121    24      36      11.81
P1      1024    30      11      0.00
=== Step 2 ===
Process Weight  Remain  Slice   vruntime
P0      3121    24      36      11.81
P1      1024    19      11      11.00
=== Step 3 ===
Process Weight  Remain  Slice   vruntime
P0      3121    24      36      11.81
P1      1024    8       11      22.00
=== Step 4 ===
Process Weight  Remain  Slice   vruntime
P0      3121    0       36      19.69
P1      1024    8       11      22.00
=== Step 5 ===
Process Weight  Remain  Slice   vruntime
P0      3121    0       36      19.69
P1      1024    0       11      30.00
=== Gantt chart ===
0 P0 36 P1 47 P1 58 P0 82 P1 90
```

The above Gantt chart string is equivalent to the following Gantt chart:

P0	P1	P1	P0	P1

0 36 47 58 82 90

Completely Fair Scheduler (CFS) Overview

In this course, you learned several scheduling algorithms. Most of them are based around the concept of fixed time slice. In contrast, CFS uses a simple counting-based technique called virtual runtime (`vruntime`).

Each process has its `vruntime`, with a default value `0`. As each process runs, it accumulates `vruntime`. When a scheduling decision occurs, CFS will pick an unfinished process with the smallest `vruntime` to run next.

CFS has the following 3 configuration strategies:
- Scheduler Latency (`sched_latency`)
- Minimum Granularity (`min_granularity`)
- Controlling the process priority

CFS Concept: Scheduler Latency (sched_latency)

CFS uses `sched_latency`, with a typical value like `48ms`, to determine how long one process should run before considering a switch. If we have `2` processes, without considering the process priority, the per-process time slice is equal to: `48/2 = 24ms`

In the later section, CFS Concept: Controlling the process priority, we will discuss how to calculate the per-process time slice when the process priority is considered.

CFS Concept: Minimum Granularity (min_granularity)

Excessive context-switch may occur if the per-process time slice is too short. If the per-process time slice is too short, CFS performance will be degraded due to the accumulated overhead of context-switch. CFS adds `min_granularity`, with a typical value like `6ms`, to control the minimum per-process time slice.

In the previous example, the per-process time slice is equal to `24ms`, so we don't need to change the per-process time slice because it is larger than `min_granularity` (`6ms`).

For example, if there are `12` processes and `sched_latency` is `48ms`. Per-process time slice is `48/12` = `4ms`, which is smaller than `min_granularity` (`6ms`). The per-process time slice will be set to `6ms`

CFS Concept: Controlling the process priority

CFS enables controls over process priority, enabling users to give some processes a higher/lower share of the CPU. The classic UNIX (i.e., the predecessor of Linux) mechanism known as the nice level is adopted.

The nice parameter can be set anywhere from `-20` to `19` for a process, with a default nice value `0`. Positive nice values imply lower priority and negative values imply higher priority.

CFS maps the nice values (defined in Unix/Linux) to the CFS weights:

- Note: The following mapping is implemented in the skeleton code

```
static const int DEFAULT_WEIGHT = 1024;
static const int NICE_TO_WEIGHT[40] = {
    88761, 71755, 56483, 46273, 36291, // nice: -20 to -16
    29154, 23254, 18705, 14949, 11916, // nice: -15 to -11
     9548,  7620,  6100,  4904,  3906, // nice: -10 to  -6
     3121,  2501,  1991,  1586,  1277, // nice:  -5 to  -1
     1024,   820,   655,   526,   423, // nice:   0 to   4
      335,   272,   215,   172,   137, // nice:   5 to   9
      110,    87,    70,    56,    45, // nice:  10 to  14
       36,    29,    23,    18,    15, // nice:  15 to  19
};
```

These weights allow us to compute the effective time slice of each process, but now accounting for their priority differences. Here is the exact formula implemented in the skeleton code:

- `weight` refers to the weighting of the process, which is looked up from the table
- `sched_latency` is the value read from the input
- `sum_of_weight` refers to the total sum of the per-process weighting

```
int calculate_per_process_time_slice(
    int weight,             // weight of a process
    int sched_latency,      // the scheduler latency
    int sum_of_weight       // total sum of weights
    ) {
    return (int)((double) weight * sched_latency / sum_of_weight);
}
```

Suppose we have the following 2 processes; the time slices are calculated at the last column of the following table:

- Note 1: `sum_of_weight` = `3121+1024` = `4145`
- Note 2: both time slices are larger than `min_granularity` (`6ms`)

Process	Burst Time	Nice Value	Weight (from table)	Time slice (calculated)
P0	60	-5	3121	36
P1	30	0	1024	11

CFS Concept: Updating vruntime

By considering the weighting, the following formula is used to update the `vruntime`. The formula implementation is provided in the skeleton code:

- `DEFAULT_WIGHT` is equal to `1024`
- `vruntime` refers to the current vruntime
- `runtime` refers to how much time the process run
- `weight` refers to the weighting of the process (see the previous section)

```
double calculate_new_vruntime(
    double vruntime, // the current vruntime
    double runtime, // how much time the process run
    double weight //  weight of a process
    ) {
    return vruntime + (double) DEFAULT_WEIGHT / weight * runtime;
}
```

Simplified CFS: How to pick the next process to run?

In each step, we need to pick an unfinished process with the smallest `vruntime` to run next. If there are more than one such processes having the same smallest `vruntime`, pick the process with the smallest process ID. For example, if both P0 and P1 have the smallest `vruntime`, we pick P0 because it has a smaller process ID.

In the Linux kernel CFS implementation, a data structure named as red-black tree should be used. Red-black tree is one of many types of balanced trees, which gives a logarithmic running time for each query.

In this project, you **DO NOT** need to implement the red-black tree data structure. In each step, you only need to search the whole list of process to find the process with the smallest `vruntime`, with the worst-case linear running time.

A Step-by-Step CFS Example

For example, we have the following 2 processes.

In the initial step (step0):

Process	Weight	Remain Time	Time slice	vruntime
P0	3121	60	36	0.00
P1	1024	30	11	0.00

Please note that 2 decimal places are used to display `vruntime`

P0 is picked to run because it has the smallest `vruntime`. Indeed, both P0 and P1 have the smallest `vruntime`, but P0 is the process having the smallest process ID. P0 runs for `36ms`, and the `vruntime` is updated based on the above formula. The table is updated as follows:

Process	Weight	Remain Time	Time slice	vruntime
P0	3121	24	36	11.81
P1	1024	30	11	0.00

P1 is picked to run because it has the smallest `vruntime`. P1 runs for `11ms`, and the `vruntime` is updated based on the above formula. The table is updated as follows:

Process	Weight	Remain Time	Time slice	vruntime
P0	3121	24	36	11.81
P1	1024	19	11	11.00

P1 is picked to run because it has the smallest `vruntime`. P1 runs for `11ms`, and the `vruntime` is updated based on the above formula. The table is updated as follows:

Process	Weight	Remain Time	Time slice	vruntime
P0	3121	24	36	11.81
P1	1024	8	11	22.00

P0 is picked to run because it has the smallest `vruntime`. P0 runs for `24ms` (Note: The remaining time is smaller than the time slice) and the `vruntime` is updated based on the above formula. The table is updated as follows:

Process	Weight	Remain Time	Time slice	vruntime
P0	3121	0	36	19.69
P1	1024	8	11	22.00

P1 is picked to run (Note: Even P0 has the smallest `vruntime`, but P0 is already finished, thus P1 is a process having the smallest `vruntime` in the current process list). P1 runs for `8ms` (Note: The remaining time is smaller than the time slice) and the `vruntime` is updated based on the above formula. The table is updated as follows:

Process	Weight	Remain Time	Time slice	vruntime
P0	3121	0	36	19.69
P1	1024	0	11	~~33.00~~ 30.00

Now, all processes are finished. The final Gantt chart is:

P0	P1	P1	P0	P1

0 36 47 58 82 90

Compilation

The following command can be used to compile the program

```
$> gcc -std=c99 -o cfs cfs.c
```

The option `c99` is used to provide a more flexible coding style (e.g., you can define an integer variable anywhere within a function)

Sample test cases

Several test cases (both input files and output files) are provided. We don't have other hidden test cases.

The grader TA will probably write a grading script to mark the test cases. Please use the Linux `diff` command to compare your output with the sample output. For example:

```
$> diff --side-by-side your-outX.txt  sample-outX.txt
```

Sample Executable

The sample executable (runnable in a CS Lab 2 machine) is provided for reference. After the file is downloaded, you need to add an execution permission bit to the file. For example:

```
$> chmod u+x cfs
```

Development Environment

CS Lab 2 is the development environment. Please use one of the following machines (`csl2wk`**XX**`.cse.ust.hk`), where **XX**=`01…40`. The grader will use the same platform.

In other words, *"my program works on my own laptop/desktop computer, but not in one of the CS Lab 2 machines*" is an invalid appeal reason. Please test your program on our development environment (not on your own desktop/laptop) thoughtfully before your submission, even you are running your own Linux OS. Remote login is supported on all CS Lab 2 machines, so you are not required to be physically present in CS Lab 2.

Project TA - Peter (cspeter@cse.ust.hk): Handling questions before the deadline
Grader TA - Kaiqiang (kxuar@cse.ust.hk): Grading and appeal handling after the deadline

Marking Scheme

1. (100%) Correctness of the 10 provided test cases. We don't have other hidden test cases, but we will check the source codes to avoid students hard coding the test cases in their programs. Example of hard coding: a student may simply detect the input and then display the corresponding output without implementing the program
2. Make sure you use the Linux diff command to check the output format.
3. Make sure to test your program in one of our CS Lab 2 machines (not your own desktop/laptop computer)
4. Please fill in your name, ITSC email, and declare that you do not copy from others. A template is already provided near the top of the source file.

***Plagiarism**: Both parties (i.e., students providing the codes and students copying the codes) will receive 0 marks. Near the end of the semester, a plagiarism detection software (MOSS) will be used to identify cheating cases. **DON'T** do any cheating!*

Submission

File to submit:

`cfs.c`

Please check carefully you submit the correct file.

In the past semesters, some students submitted the executable file instead of the source file. Zero marks will be given as the grader cannot grade the executable file.

You are not required to submit other files, such as the input test cases.

Late Submission

For late submission, please submit it via email to the grader TA.
There is a 10% deduction, and only 1 day late is allowed
(Reference: Chapter 1 of the lecture notes)

References

This project is modified based on the discussion of CFS in Chapter 9 - Scheduling: Proportional Share of **Operating Systems: Three Easy Pieces**
Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau
Arpaci-Dusseau Books
August, 2018 (Version 1.00)
Free book chapters are available: https://pages.cs.wisc.edu/~remzi/OSTEP/#book-chapters