COMP3511

Part 1: Project Assignment 2 Introduction

Introduction

- The Linux kernel implements a completely fair scheduler (CFS)
- In this project, we need to implement a simplified version of CFS
- Please note that the details of CFS are not covered in the lecture notes
- In this lab, we are going to cover the detailed requirements of implementing a simplified CFS

How to run the program?

- Here is a sample usage:

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

`$>` represents the shell prompt

`<` means input redirection

`>` means output redirection

- Thus, you can easily replace `input.txt` with different test cases and then use the `diff` command to compare with the sample output files

Skeleton code

- The input parsing is given in the skeleton code
- 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
  - Please read the skeleton code carefully

Sample input and output

Sample Input

```
# 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
```

Sample Output

```
=== 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
```

Input format

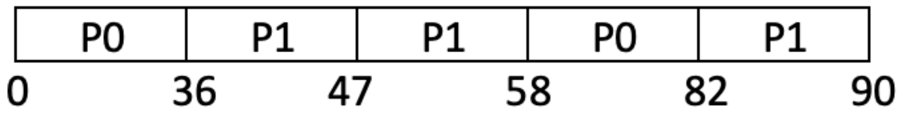
- The input parsing is given in the skeleton code
- Empty lines and lines starting with # are ignored
- Format of constant: `name = <value>`
- Format of vector: `name = <values of the vector>`

```
# 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
```

Sample Input

Output format

- The output consists of 3 regions:
  1. Display the parsed values
  2. Display the intermediate steps
  3. Display the final Gantt chart
- The final Gantt chart string is equivalent to:



Sample Output

```
=== 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
```

1st region

2nd region

3rd region

Completely Fair Scheduler (CFS) Overview

- 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 <u>smallest</u> `vruntime` to run next

CFS Configuration Strategies

- Scheduler Latency (`sched_latency`)
- Minimum Granularity (`min_granularity`)
- Controlling the process priority

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
- Example:
  - If we have `2` processes, without considering the process priority, the per-process time slice is equal to: `48/2 = 24ms`
- We will discuss how to calculate the per-process time slice when the process priority is considered

Minimum Granularity (`min_granularity`)

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

Controlling the process priority

- 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.

Mapping Nice Values to CFS Weights

- CFS maps the nice values (defined in Unix/Linux) to the CFS weights:
  - 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
};
```

Calculating the per-process time slice

- 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

```
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);
}
```

Example: Calculating the per-process time slice

- 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

Why scaling the time slices? Reason: If all time slices are larger than `min_granularity`, the sum of all time slices should be roughly equal to `sched_latency`

Updating vruntime

- The following formula is used to update the `vruntime`
  - Note: The formula implementation is provided in the skeleton code:

```
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
- What happen if we have more than one choices?
  - If there are more than one 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

Simplified CFS: Any special data structure?

- 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 **<u>DO NOT</u>** 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, suppose we have the following 2 processes.
  - Please note that 2 decimal places are shown for the current `vruntime`:
- Step 0:
  - Question: Which process will be picked next?

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

Step 1

- 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`
- The table is updated as follows:
  - Question: Which process will be picked next?

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

Step 2

- P1 is picked to run because it has the smallest `vruntime`
- P1 runs for `11ms`
- The table is updated as follows:
  - Question: Which process will be picked next?

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

Step 3

- P1 is picked to run because it has the smallest `vruntime`
- P1 runs for `11ms`
- The table is updated as follows:
  - Question: Which process will be picked next?

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

Step 4

- 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
- The table is updated as follows:
  - Question: Which process will be picked next?

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

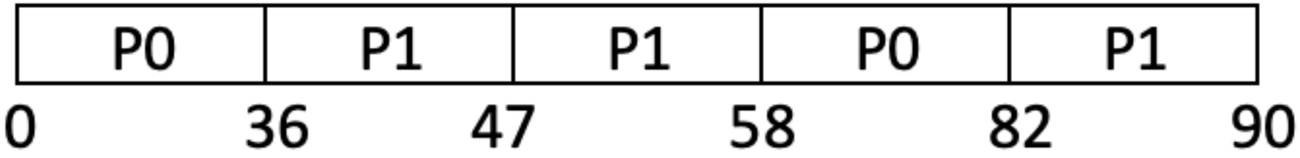
Step 5

- P1 is picked to run
  - Note: Even P0 has the smallest `vruntime`, but it 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
- The table is updated as follows:
  - Question: Which process will be picked next?

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

The final Gantt Chart

- No processes can be picked next because all processes are finished
  - `Remain Time` = `0` for all processes
- The final Gantt chart is:



Sample test cases

- Test cases are provided
- 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

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

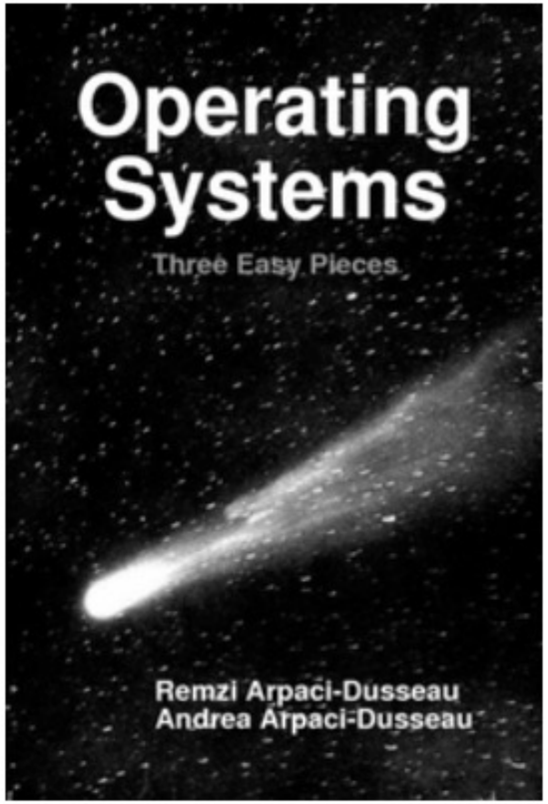
  - An extra option `--suppress-common-lines` can be added if you are not interested in the common lines. If both text files are the same, adding `--suppress-common-lines` will print nothing on the screen.

Summary

- **Think** carefully before you type **ANY** line of code
  - Good C programmers never do trial-and-error
  - A program that can compile does not mean that it can execute correctly
  - Check carefully to avoid runtime errors (i.e., Segmentation fault)
- Read carefully the provided base code
- Compare your output files with the sample output files using the Linux diff command

References

- This project is modified based on the discussion of CFS in Chapter 9 - Scheduling: Proportional Share of Operating Systems: Three Easy Pieces
- This book is one of the reference books in this course
- Free book chapters are available: https://pages.cs.wisc.edu/~remzi/OSTEP/#book-chapters



Live Demo

The skeleton code

The sample Linux executable program