OS - Project 1 - Report

b07902131

1 Design

1.1 System calls

1.1.1 `osproj1_gettime`

Receives a `struct timespec` pointer, and stores time info by `getnstimeofday` into the pointer.

1.1.2 `osproj1_pinfo`

Receives 5 parameter, 1 with `pid`, 2 with `start_time` and last 2 with `end_time`. Kernel message level is set to `KERN_INFO`.

1.2 `util`

This file includes some function that may used by multiple part of codes. `SetProcessCPU` for setting the CPU affinity for given `pid`, `SetPriority` set the priority for a `pid`, and `StartProcess` will fork another process and run `TIME_UNIT` for a specified times.

Before the forked child process starts and after finishs running its `TIME_UNIT`, it will record its starting time from the system call `osproj_gettime`, and print them just before its exit via `osproj_pinfo`.

In most cases, the running process will be set to priority 99, and other process is set to priority 1. The affinity CPU of these processes will be set to CPU1.

In `struct Process`, there is a variable `spawned` records whether the process is spawned. In SJF and PSJF, there is a chance a process starts running even if there are other process must to run first since the shorter process may be spawned later than the longer one, and the longer one will be context switch by CPU even if its priority is 1 since there are no other priority 99 process. To avoid this, in SJF and PSJF, the process will not actually fork once the process is ready but wait until they are scheduled by scheduler at the first time, thus ensure the start time of the process is currect.

1.3 `meow`

A implement of self-invented self-balanced binary search tree. Only `Insert`, `PeekBegin`, `PopBegin` are implemented, with expected complexity $O(\log N)$ each, where N is the current tree size.

I will refer this tree as `meow` in below sections.

1.4 `scheduler`

This file is the entry point of the whole process. This file will be responsible for reading the input (the description of the processes), sort them by their start time, set its affinity CPU to CPU0 and call the corresponding scheduler method according to the input.

The sorting comparision function will consider the ready time as the first key and original input order as the second key, from less to greater. This definition is at `util`.

1.5 `FIFO`

Use two pointer to identify ”which process is ready to execute (pushed into queue)” and ”which process is scheduled to run”. When all job is done (the second pointer is located at `n - 1` (`n` is the number of processes, given in the input) and that job is over), finish this scheduler.

When each job is done, `wait` the forked process (this is same for all schedulers). At the begin of each time unit, try moving the first pointer to find if there is a process can be started (forked), and if there are a queueing process and CPU is empty, start a process (adjust priority of the process to 99).

1.6 `RR`

The basic structure of RR scheduler is same as FIFO. But we need to add a queue to maintain the order of unfinished but started jobs. The prototype of `struct Queue` is defined in `RR.h`.

For a process that runs for `kRRRound` (which is 500 by the project definition) continuous time units), it will be moved to the end of the queue and wait for execute again (the priority will, obviously, be changed to 1). For a new ready job, it will be spawned and push into the end of the queue too. In my implementation, for a process A that runs for `kRRRound` time units and a process B newly spawned in same time unit, B will be in the front of A inside the queue.

1.7 `SJF`

The basic structure is still same as FIFO and RR. `meow` is used to store the processes in order of their running time. When there is no process is running, the first (with smallest running time) will be pop out from `meow` and start executing. Just like what I just mentioned in `util` section, the process will not be spawned instantly right after it is ready but after it is scheduled.

1.8 PSJF

The structure is same as SJF, except that when every new process is ready, we have to check whether the execution time is less than the process that is currently running. If yes, then we have to switch these two process (inserting the running one into `meow`, and pop out the shortest process from it).

2 Kernel Version

4.14.175 on Ubuntu 18.04.4 LTS.

3 Comparison

All results is stores in `output`. Run `check.js` will print all timing result and comparing result with theoretical output. The result is printed in `check.result`.

Comparing to the theoretical result generated by `check.js`, the most of differences between real output and theoretical result is around 5% with small variation. This difference is quite reasonable.

In `TIME_MEASUREMENT`, all processes can start instantly after it is ready, thus the scheduler is the main part contributing to the time whole process spends (wait for ”time” in scheduler to make the process ready). In most of other testcases, there are other processes queueing to be scheduled, thus the time is based on the forked processes but not scheduler (the process will be ready first. Once the previous process is done, next process can execute directly without waiting scheduler). We know that the scheduler does much more (fork, more system calls to set priority, etc.) than forked process (simply runs for loops), which leads to more slow executing speed. This is a possibly explanation of this result.

Also, there exist other processes in the vm, which may leads to actual context switch by CPUs, leads to more error in timing.