OS Project #1

B07902024

April 29th, 2020

1 Design

1.1 `src/process.h`

The structure of a process is defined in this file. The function `ForkProcess` forks and executes the job program. To make sure that all the processes excluding the main scheduler are running on the same core, once the new process is forked, the CPU on which it is allowed to run is set to the first core by the system call `sched_setaffinity`. In addition to affinity, the priority of the process is set to the lowest possible (cf. `RunProcess` and `PauseProcess`) so that it will only be ran if the scheduler says so in the future.

Two of the main utility function is `RunProcess` and `PauseProcess`, which starts or resumes the process by setting its priority to the highest possible and pause the running process by lowering its priority, respectively. The highest and lowest possible priority is obtained by the system call `sched_get_priority_max` and `sched_get_priority_max`. The policies of all processes is set to `SCHED_FIFO` in the kernel.

Each forked process is associated with a status being either `WAITING` or `RUNNING`, indicating whether the process is now running (with highest priority) or waiting (with lowest priority). At most one process will be in running state at a time.

1.2 `src/queue.h`, `src/vector.h`, `src/heap.h`

Since there is no standard library of these basis data structures in C, the simple implementations are provided in these files. `Queue` is used in both `FIFO` and `RR` policy, which supports FIFO access of the elements.

`Vector` is a dynamic container of processes which supports random access of elements, pusing and popping elements from the back. To achieve $O(1)$ amortized complexity, the allocation only happens when the capacity of the vector runs out, and the capacity is doubled then.

`Heap` is implemented on top of `Vector` and is used in policy `SJF` and `PSJF`. `Heap` supports inserting process, querying and deleting process with the smallest remaining time in $O(\log N)$ time.

1.3 `src/schedule.h`

The file contains the implementation of the main scheduling algorithms of the four policies. The processes to be scheduled are sorted in ascending order of the starting tiems.

1.3.1 First-in First-out (FIFO)

A `Queue` containing all active processes is maintained, and in each time unit t, all processes starting at time t is first forked then pushed into the queue. All processes in the queue except the first one are in the waiting state. Each time we check if the first process is completed, and if so, we discard it from the queue and runs the next process using `StartProcess`.

Forking processes that start at time t and removing completed process are the first thing to be done in each time unit in all scheduling algorithms, and will be omitted in the following descriptions.

1.3.2 Round Robin (RR)

Similar to `FIFO`, we maintain all active processes in a queue. The difference is that the first element is not the only process that had been run: some processes may have run before but are now paused because of the policy. If the first process in the queue is now in the running state and has been running for `kRoundRobin = 500` time units, it is moved from the beginning of the queue to the end of the queue to let the next process run.

1.4 Shortest Job First (SJF)

Since we need to find the active process with the shortest execution time, processes should be kept in a heap instead of a queue. A pointer `running` is pointed to the currently running process, and in each time unit we should pop the smallest element p out of the heap and point `running` to p if the currently running process is completed.

1.5 Preemptive Shortest Job First (PSJF)

Instead of keeping the pointer `running` run until it is completed, since preemption is allowed in PSJF, if the newly added process p has smaller remaining time, we should pause `running`, push `running` back into the heap and replace `running` with p. Implementation-wise, the currently running process is not discard from the heap. Instead, since the running process will always has the smallest remaining time unless some new processes are added, we can directly decrement the remaining time of it without popping it from the heap. If at some time we find out that the smallest element in the heap is in the waiting state, we know that `running` is preempted and thus we should pause `running` and runs the shortest process.

2 Kernel Version

The current stable release of Linux Kernel (version 5.6.7) is used. Note that the definition of system calls in version 5.6.7 is different from version 4.x (as in homework #1), so the kernel files may not compile in older version. In particular, instead of directly declaring the system calls like:

```
asmlinkage long a_plus_b(int a, int b) {
  return a + b;
}
```

one should wrap the definition in the macro:

```
SYSCALL_DEFINE2(a_plus_b, int, a, int, b) {
  return a + b;
}
```

2.1 Kernel files

Because `getnstimeofday` seems to be deprecated in the newer version of Linux Kernel, `ktime_get_ts64` [1] (which returns the time elapsed since boot in nanosecond precision) is used instread. The new system calls `gettime` and `printtime` have ID 335 and 336, respectively.

3 Results and Comparisons

To compare the actual results with the theoretical ones, the relative errors of execution times are computed. That is, let a and b be the actual execution time and the expected execution time, respectively, the relative error is defined as $\frac{|a-b|}{b}$. Then the average relative errors for each policy is listed as:

FIFO	RR	SJF	PSJF
0.016862	1.873561	0.205391	0.315899

Table 1: The relative errors between the *actual running time* and the expected one. The result can be reproduced by `benchmark.py`

Round-Robin has the largeset error among all four policies, which is because the *actual execution time* is measured as the difference between the time when the process is first executed and the time when the process terminates. Under Round-Robin policy, processes have high chances to be preempted multiple times, which results in the large error.

First-in First-out has the smallest error since the theoretical error should be 0 because no process should be preempted after it starts running. However, since the scheduling priority is set *after* the process is forked, it may happens that the child process runs before the parent process pauses it. Context-switch time may be an important factor of the error as well.

The effect by the *pause-after-fork* becomes larger in Shortest Job First policy, in which process may be delayed for a very long time due to its long execution time, comparing to the FIFO case. The error in PSJF is slightly larger than that in SJF, but much smaller in comparison with RR since processes only have to wait for shorter processes to finish in PSJF, while in RR they will be preempted by all other running processes.

In conclusion, the relative error calculated above is only meaningful in FIFO since preemption is not taken into account in other cases. Therefore, the actual running time differs from the expected running time by about 1.5%, which should be quite acceptable considering the context-switch time and other overheads.

[1] `https://www.kernel.org/doc/html/latest/core-api/timekeeping.html`