Operating Systems Project 1

Huang, Yu-Shiuan
Department of Computer Science and Information Engineering
National Taiwan University
b07902134@ntu.edu.tw

I. Environment

- Kernel Version: Linux 5.6.4-rt3 (`PREEMPT_RT` patched, `CONFIG_NO_HZ_FULL` enabled)
  - Built from https://aur.archlinux.org/packages/linux-rt/ commit `8886d513d3ef12894defd076f40179a01db5d5a6`
- Cmdline: `quiet loglevel=3 rd.systemd.show_status=false rd.udev.log-priority=3 scsi_mod.use_blk_mq=y dm_mod.use_blk_mq=y nowatchdog mitigations=off isolcpus=nohz,domain,managed_irq,1-2 irqaffinity=0,3-5 maxcpus=6 tsc=nowatchdog`
- CPU: Intel i7-8750H
- Hyperthreading disabled
- `system-tuning.sh`
  - Turbo boost disabled
  - CPU idle states disabled
  - TLP configuration: https://gitlab.com/snippets/1971598
- Scheduler compiled with PGO (`*.gcda` files attached)
- `mlockall` called on scheduler start to avoid spikes caused by page faults
- ASLR *not* disabled due to the variance betweens runs not being as significant

II. Design

A. Scheduler

For time calculation, instead of running the empty loop every tick in the scheduler, we first measure the wall time per unit time so that we can convert the former to the latter. This should be better for accuracy, i.e., not only should the timing not drift as much, but other work done in the scheduler will also not affect the timing.

Essentially, our scheduler busy-waits[1] w.r.t. the unit time for events such as the ready or preemption of processes.

Notably, due to jitters in the timing (discussed in detail in the next section), a constant ϵ is introduced so that a process is not preempted if it only requires about ϵ more units of time. Also, if we find that another process will be ready in ϵ units, we wait for the process to enqueue instead of possibly prematurely preempting other processes.

For the scheduler and running child processes, `SCHED_FIFO` with the maximum priority is used, while `SCHED_IDLE` with the lowest priority is used for preempted child processes. In addition, the scheduler and child processes are pinned to different CPU cores, namely 2 and 1 respectively.

In the following sections, each scheduling policy is discussed.

1) FIFO: For the FIFO policy, we simply wait for a process to be ready, run and wait for it, then move on to the next one.

2) RR: In the round-robin policy, we maintain a doubly-linked-list containing possibly active processes. In the event loop, we check if there are processes in the event queue that can be removed (polled via `waitpid`) and if there are processes that are ready and can be inserted. Notably, we always insert to the end of the list. Then, if the current process is dead or needs to be preempted, we search for the next process in the list and switch to it.

3) SJF: In the case of SJF, we maintain a min-heap[2], namely a leftist tree, that contains the processes that are waiting to be executed. We then simply choose the minimum element from the heap to run, while making sure that other ready processes are added to the heap when the process is running.

4) PSJF: The design of PSJF is similar to SJF, except that we need to maintain how much work a process has done to derive its remaining time and preempt processes based on that. Namely, each time an element is inserted to the heap, we check if it is able to preempt the current process, and if so, insert the latter into the heap so that we can start the former.

B. Kernel

Two system calls are implemented: `osproj_gettime` and `osproj_printk`. The former allows for querying the current time (`CLOCK_MONOTONIC`), while the latter accepts a name string and begin time; and `printk`s the name, begin time, and current time according to the assignment specifications.

Notably, although we have `osproj_gettime` and use it in the child processes, `clock_gettime` is still widely used in the scheduler. This is because `clock_gettime` is in the VDSO, and should have lower calling overhead.

III. Comparison

[1]While this pegs the CPU, so does running empty loops; hence nothing is lost here.

[2]Tie broken by the order of the processes in the input file.

Table I. FIFO_1 Comparison (E: Experiment; T: Theory)

	Start (E)	End (E)	Start (T)	End (T)
P1	0	501	0	500
P2	501	1002	500	1000
P3	1002	1503	1000	1500
P4	1503	2004	1500	2000
P5	2004	2506	2000	2500

Table II. PSJF_2 Comparison (E: Experiment; T: Theory)

	Start (E)	End (E)	Start (T)	End (T)
P1	0	4003	0	4000
P2	1000	2000	1000	2000
P3	4003	10999	4000	11000
P4	5000	6997	5000	7000
P5	7000	7996	7000	8000

Table III. RR_3 Comparison (E: Experiment; T: Theory)

	Start (E)	End (E)	Start (T)	End (T)
P1	1200	20210	1200	20200
P2	2700	20724	2700	20700
P3	4200	18199	4200	18200
P4	6200	31199	6200	31200
P5	6700	30207	6700	30200
P6	7200	28214	7200	28200

Table IV. SJF_4 Comparison (E: Experiment; T: Theory)

	Start (E)	End (E)	Start (T)	End (T)
P1	0	3002	0	3000
P2	3002	4005	3000	4000
P3	4005	8016	4000	8000
P4	9016	11023	9000	11000
P5	8016	9016	8000	9000

(Note that the unit times here are converted to using the initial measurement mentioned in II.)

(The data shown here can be found in `demo/demo-out-old`. Note that this is not the same as the data in the demo video, i.e., those in `demo/demo-out`. This is due to some video encoding issues when recording the first attempt.)

In the examples presented, the orders in which the processes are executed are correct. However, as can be seen from the FIFO case, there is still a bit of jitter in terms of the running times for the child processes. It is extremely likely that the deviations in the other cases stems from the same cause. This is despite that fact that we strived to minimize such jitter via the effort shown in I.

Unfortunately, in modern x86_64 processors, there appears to be a lot of factors at play for these variances, from microscopic factors such as cache and branch misses to macroscopic factors such as SMI interrupts.[3] That being said, it remains to be investigated what is the major cause in our case.

In addition, it may be worth exploring whether moving idle tasks to another CPU helps with the jitter, given the current Linux implementation of adaptive-ticks, i.e., `NO_HZ_FULL`, does not kick in when multiple `SCHED_FIFO` processes are running on the same core.[4]

[3] Examples are only for illustrative purposes and are not necessarily the cause for our jitter.

[4] C.f. `Documentation/timers/NO_HZ.txt` in the Linux kernel.