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`
  - `watchdog` disabled
  - Meltdown/Spectre-esque mitigations disabled
  - CPU 1 and 2 isolated and excluded from the IRQ affinity mask
  - Hyperthreading disabled
- CPU: Intel i7-8750H
- `system-tuning.sh`
  - Turbo boost disabled
  - CPU idle states disabled
  - TLP configuration: https://gitlab.com/snippets/1971598
- `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 the 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 know 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 active 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 dead processes in the list that can be removed (polled via `waitpid`) and if there are processes that are ready and can be launched and inserted to the list[2], namely, at the position before the current task. 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[3], namely a leftist tree (, which supports `DELETE_MIN` operations in $\mathcal{O}(\log N)$ time and `FIND_MIN` queries in $\mathcal{O}(1)$ time), 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 a 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 back 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 (PID in this case) and begin time; and `printk`s the name, begin time, and current time (as the syscall is invoked when before ending child processes, this is the same as the end time) according to the assignment specifications.

[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.

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

Notably, although we have `osproj_gettime` and do 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.

In addition, because of the deprecation of functions like `getnstimeofday` in newer versions of Linux (due to the 2038 problem), `ktime_get_ts64` is used instead.

III. Comparison

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

	Start (E)	End (E)	Start (T)	End (T)	$\Delta E/\Delta T$
P1	0	500	0	500	1.000
P2	500	1001	500	1000	1.002
P3	1001	1501	1000	1500	1.000
P4	1501	2001	1500	2000	1.000
P5	2002	2501	2000	2500	0.998

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

	Start (E)	End (E)	Start (T)	End (T)	$\Delta E/\Delta T$
P1	0	4002	0	4000	1.001
P2	1000	2000	1000	2000	1.000
P3	4002	10998	4000	11000	0.999
P4	5000	6997	5000	7000	0.999
P5	7000	7999	7000	8000	0.999

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

	Start (E)	End (E)	Start (T)	End (T)	$\Delta E/\Delta T$
P1	1200	19687	1200	19700	0.999
P2	2700	20175	2700	20200	0.999
P3	4200	18192	4200	18200	0.999
P4	6200	31131	6200	31200	0.997
P5	6700	30149	6700	30200	0.998
P6	8200	28163	8200	28200	0.998

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

	Start (E)	End (E)	Start (T)	End (T)	$\Delta E/\Delta T$
P1	0	2995	0	3000	0.998
P2	2995	3994	3000	4000	0.999
P3	3994	7990	4000	8000	0.999
P4	8988	10984	9000	11000	0.998
P5	7990	8988	8000	9000	0.998

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

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 the fact that we strived to minimize such jitter via the effort shown in I.

Unfortunately, not only does OS-caused interrupts cause such deviations, but in modern x86_64 processors there also appears to be a lot of factors at play, from microscopic factors such as cache and branch prediction to macroscopic factors such as SMI interrupts.[4] That being said, it remains to be investigated what the major cause is 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.[5]

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

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