Operating System Project 2

Advanced Scheduler

20195008 곽병혁, 20195052 김희수, 20195182 최승훈

April 6, 2024

1 Definition

scheduler maximizes the usage of a processor by process scheduling. While the scheduler works on scheduling the order of executing processes is crucial since each processor can execute a process one by one and the process does not always occupy processors. Several process scheduling methods have been introduced such as First Come First Serve (FIFS) Scheduling and Short Job First (SJF) Scheduling. In the case of PintOS, a Round-Robin approach is used for the scheduling. A Round-Robin allocates process without priority so that a first-arrived process is executed first. This strategy can shorten the response time. However, it requires frequent context switching, which yields significant overhead. A convoy effect, the symptom caused by the long turnaround time process, can also occur. Thus, we desire to convert Round-Robin into priority scheduling. Priority scheduling can be the answer to both problems in Round-Robin. Whereas, this priority scheduling results in starvation, a certain process waits for infinite time due to the low priority. this drawback can be solved by aging.

The other problem with priority scheduling is priority donation. When several threads are waiting for the lock that a low-priority thread has, either sequential or parallel way, priorities inverse. That means high-priority processes wait for the termination of a low-priority process. This priority inversion does not align with our priority policy so it needs to be handled manually.

2 Policy

2.1 Priority Scheduling

Our strategy to implement priority scheduling is pushing processes into the *ready_list* on the order of priority so that a thread in the *ready_list* with the highest priority can be always selected to run. Three functions (*thread_yield()*, *thread_unblock()*, *thread_create()*), are towards THREAD_READY as shown in **Figure 1** so we modify the way those functions push the process. Furthermore, the processor should yield immediately to a new thread which has higher priority than the running thread and is just added to the *ready_list*. Lastly, threads waiting on a semaphore, lock, or condition variable should have the highest priority.

2.2 Priority Donation

There are briefly two different types of priority donation; multiple donation and nested notation. Multiple donation means a situation in which threads are waiting for a certain thread that is holding the lock. On the other hand, nested donation occurs when threads are sequentially waiting for the lock. To handle these donations, lock dependency needs to be checked whenever some process gets or releases the lock. Lock dependency means which process is holding the lock and which process is waiting for it. If a certain process with high-priority requests to get the occupied lock, the high-priority process shares its priority with a lock holder process so that such a task can be done faster. After finishing the task using the lock, the high-priority process should return the lock to the original lock holder process and then that holder process restore its priority.

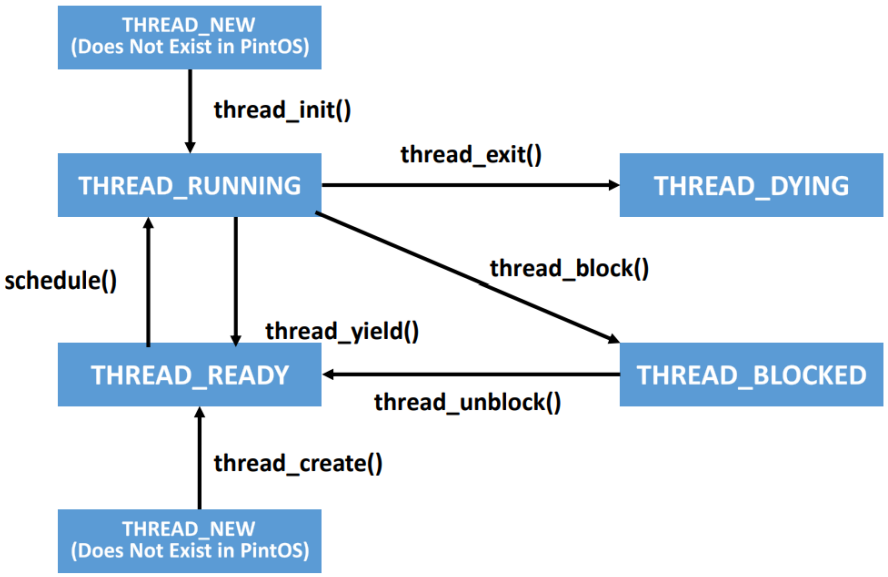

Figure 1: Process life cycle in PintOS.

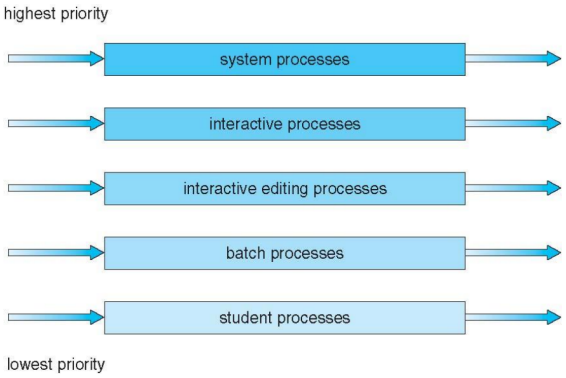

Figure 2: Multilevel queue scheduling scheme.

2.3 Advanced Scheduler

Now implement an advanced scheduler called 4.4BSD Scheduler for balancing threads' requests from CPU-bound threads and I/O-bound threads. The shared reference[1] illustrated the details we should implement. The new scheduler manages multi-level queues with different priorities as shown in **Figure 2**. By defining 'niceness' and a new formula for the priority, the scheduler selects a proper thread from the non-empty highest thread queue for every tick. Niceness refers to how "nice" this thread is; tends to yield one's CPU time to the other threads. 4.4BSD scheduler takes how much the thread recently used CPU into account so that ensures all threads can access the CPU and prevents starvation.

3 Algorithm

3.1 Priority Scheduling

As mentioned above, the system pushes new threads in *ready_list* with their priority so that the new scheduler executes threads following the high-priority order. There are only two paths where *elem* is pushed to *ready_list*; *thread_unblock()* and *thread_yield* are revised. *thread_create()*

[1] https://web.stanford.edu/class/cs140/projects/pintos/pintos_7.html#SEC131

seems like another path as illustrated in **Figure 1** but it calls *thread_unblock()* so no need to consider. Beyond that, the scheduler compares the priority between the currently running thread on the processor and a new thread that has just been added right after the insertion to give way to the new thread immediately. *thread_yield()* automatically yields the processor since it calls *schedule()*. Functions that call *thread_unblock()* such as *thread_create()*, however, demand additional priority comparison. Hence such functions are modified. While implementing this new scheduler, all semaphore, lock, or condition variables get the highest priority by calling new *thread_unblock()* in *sema_up()*.

3.2 Priority Donation

The fundamental task to deal with donation is adding member variables for saving and restoring threads' priority level in *thread* structure. Then, priority comparing, restoring, setting, and donating functions with new parameters are added so one can handle the donation. Such a priority donating procedure involves lock acquisition and release but is not implemented in the original scheduler. Hence functions to deal with lock are needed. The nested donation problem is easy to resolve if the scheduler is capable of untangling the multiple donation situation. Iterating priority donation is sufficient. Note that semaphore and condition variables also have priorities we should take priority donation from them into account. The order of allocating shared resources to semaphore and condition variables depends on *waiter* value so *waiter* insertion also reflects the priority.

3.3 Advanced Scheduler

First and foremost, a variable estimating the expected number of threads ready to run within a minute, *load_avg*, applies to the calculation of *recent_cpu* to measure how long a thread uses CPU. Both variables are updated every second by **Equation 1, 2**. This *recent_cpu* adopted an exponentially weighted moving average to compute it.

$$\text{load\_avg} \leftarrow \frac{59}{60} \times \text{load\_avg} + \frac{1}{60} \times \text{ready\_threads} \tag{1}$$

$$\text{recent\_cpu} \leftarrow \frac{2 \times \text{load\_avg}}{2 \times \text{load\_avg} + 1} \times \text{recent\_cpu} + \text{nice} \tag{2}$$

Next, the scheduler reproduces the priorities of all threads every 4 ticks by the **equation 3**.

$$\text{priority} = \text{PRI\_MAX} - \frac{\text{recent\_cpu}}{4} - 2 \times \text{nice} \tag{3}$$

Through this procedure, queues that the advanced scheduler become multi-level feedback queues. Now threads that have low priority can be executed without long waiting.

4 Mechanism

4.1 Priority Scheduling

[Modified]

- thread_unblock() in thread.c
- thread_yield() in thread.c
- thread_create() in thread.c
- sema_up() in synch.c
- signal() in intq.c

[Implemented]

- elem_compare_priority() in thread.c
- check_high_priority() in thread.c

4.2 Priority Donation

[Modified]

- thread in thread.c
- init_thread() in thread.c
- thread_get_priority() in thread.c
- donate_priority() in thread.c
- lock_acquire() in synch.c
- lock_release() in synch.c
- sema_down() in synch.c
- cond_wait() in synch.c
- cond_signal() in synch.c
- sema_compare_priority() in synch.c

[Implemented]

- compare_donation_priority() in thread.c
- renew_priority() in thread.c
- donate_priority() in thread.c
- thread_set_priority() in thread.c

4.3 Advanced Scheduler

[Modified]

- thread_set_nice() in thread.c
- thread_get_nice() in thread.c
- thread_get_load_avg() in thread.c
- thread_get_recent_cpu() in thread.c
- timer_interrupt() in timer.c
- lock_acquire() in synch.c
- lock_release() in synch.c
- thread_set_pririty() in thread.c

[Implemented]

- thread/fixed_point.h
- calc_mlfqs_pri() in thread.c
- calc_mlfqs_recent_cpu() in thread.c
- calc_mlfqs_load_avg() in thread.c
- up_mlfqs_recent_cpu() in thread.c
- recalc_mlfqs_recent_cpu() in thread.c
- recalc_mlfqs_pri() in thread.c

5 Result

```
pass tests/threads/mlfqs-block
pass tests/threads/alarm-single
pass tests/threads/alarm-multiple
pass tests/threads/alarm-simultaneous
pass tests/threads/alarm-priority
pass tests/threads/alarm-zero
pass tests/threads/alarm-negative
pass tests/threads/priority-change
pass tests/threads/priority-donate-one
pass tests/threads/priority-donate-multiple
pass tests/threads/priority-donate-multiple2
pass tests/threads/priority-donate-nest
pass tests/threads/priority-donate-sema
pass tests/threads/priority-donate-lower
pass tests/threads/priority-fifo
pass tests/threads/priority-preempt
pass tests/threads/priority-sema
pass tests/threads/priority-condvar
pass tests/threads/priority-donate-chain
pass tests/threads/mlfqs-load-1
pass tests/threads/mlfqs-load-60
pass tests/threads/mlfqs-load-avg
pass tests/threads/mlfqs-recent-1
pass tests/threads/mlfqs-fair-2
pass tests/threads/mlfqs-fair-20
pass tests/threads/mlfqs-nice-2
pass tests/threads/mlfqs-nice-10
pass tests/threads/mlfqs-block
All 27 tests passed.
make[1]: Leaving directory `/home/aiden/pintos/src/threads/build'
aiden@ubuntu:~/pintos/src/threads$
```

Figure 3: Final results of 27 *make check* tests.

Acknowledgment All source codes are available in https://github.com/Aiden-Kwak/pintos2_p2_backup.