Section 6: Scheduling, Deadlock

CS 162

March 5, 2021

Contents

1 Vocabulary

- **Scheduler** - Routine in the kernel that picks which thread to run next given a vacant CPU and a ready queue of unblocked threads.
- **Linux CFS** - Linux scheduling algorithm designed to optimize for fairness. It gives each thread a weighted share of some target latency and then ensures that each thread receives that much virtual CPU time in its scheduling decisions.
- **Multi-Level Feedback Queue Scheduling** - MLFQS uses multiple queues with priorities, dropping CPU-bound jobs that consume their entire quanta into lower-priority queues.
- **Priority Inversion** - If a higher priority thread is blocking on a resource (a lock, as far as you're concerned but it could be the Disk or other I/O device in practice) that a lower priority thread holds exclusive access to, the priorities are said to be inverted. The higher priority thread cannot continue until the lower priority thread releases the resource. This can be amended by implementing priority donation.
- **Priority Donation** - If a thread attempts to acquire a resource (lock) that is currently being held, it donates its effective priority to the holder of that resource. This must be done recursively until a thread holding no locks is found, even if the current thread has a lower priority than the current resource holder. (Think about what would happen if you didn't do this and a third thread with higher priority than either of the two current ones donates to the original donor.) Each thread's effective priority becomes the max of all donated priorities and its original priority.
- **Deadlock** - A case of starvation due to a cycle of waiting. Computer programs sharing the same resource effectively prevent each other from accessing the resource, causing both programs to cease to make progress.
- **Banker's Algorithm** - A resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, before deciding whether allocation should be allowed to continue.

2 Scheduling

2.1 Simple Priority Scheduler

We are going to implement a new scheduler in Pintos we will call it SPS. We will just split threads into two priorities "high" and "low". High priority threads should always be scheduled before low priority threads. Turns out we can do this without expensive list operations.

For this question make the following assumptions:

- Priority Scheduling is NOT implemented
- High priority threads will have priority 1
- Low priority threads will have priority 0
- The priorities are set correctly and will never be less than 0 or greater than 1
- The priority of the thread can be accessed in the field `int priority` in `struct thread`
- The scheduler treats the ready queue like a FIFO queue
- Dont worry about pre-emption.

Modify `thread_unblock` so SPS works correctly.

You are not allowed to use any non constant time list operations

```
void
thread_unblock (struct thread *t)
{
  enum intr_level old_level;
  ASSERT (is_thread (t));
  old_level = intr_disable ();
  ASSERT (t->status == THREAD_BLOCKED);

  ------------------------------------------------

  ------------------------------------------------

  ------------------------------------------------

  ------------------------------------------------
  list_push_back (&ready_list, &t->elem);

  -------------------------------------------------
  t->status = THREAD_READY;
  intr_set_level (old_level);
}
```

2.1.1 Fairness

In order for this scheduler to be ”fair” briefly describe when you would make a thread high priority and when you would make a thread low priority.

2.1.2 Better than Priority Scheduler?

If we let the user set the priorities of this scheduler with `set_priority`, why might this scheduler be preferable to the normal pintos priority scheduler?

2.1.3 Tradeoff

How can we trade off between the coarse granularity of SPS and the super fine granularity of normal priority scheduling? (Assuming we still want this fast insert)

3 Deadlock

3.1 Introduction

What are the four requirements for Deadlock?

What is starvation and what is deadlock? How are they different?

3.2 Banker's Algorithm

Suppose we have the following resources: A, B, C and threads T1, T2, T3 and T4. The total number of each resource as well as the current/max allocations for each thread are as follows:

Total		
A	B	C
7	8	9

	Current			Max		
T/R	A	B	C	A	B	C
T1	0	2	2	4	3	3
T2	2	2	1	3	6	9
T3	3	0	4	3	1	5
T4	1	3	1	3	3	4

Is the system in a safe state? If so, show a non-blocking sequence of thread executions.

Repeat the previous question if the total number of C instances is 8 instead of 9.

4 Fall 2019 Practice Midterm

4.1 Optional Practice: Delivery Service

Assume each numbered line of code takes 1 CPU cycle to run, and that a context switch takes 2 CPU cycles. Hardware preemption occurs every 50 CPU cycles and takes 1 CPU cycle. The scheduler is run after every hardware preemption and takes 0 time. Finally, the currently running thread does not change until the end of a context switch.

```
Lock lock_a, lock_b; // Assume these locks are already initialized and unlocked.
int a = 0;
int b = 1;
bool run = true;

    Kiki() {
1.      bool cond = run;
2.      while (cond) {
3.          int x = a;
4.          int y = b;
5.          int sum = x + y;
6.          lock_a.acquire();
7.          lock_b.acquire();
8.          a = y;
9.          b = sum;
10.         lock_a.release();
11.         lock_b.release();
12.         cond = run;
        }
    }
    Jiji() {
1.      bool cond = run;
2.      while (cond) {
3.          int x = b;
4.          int sum = a + b;
5.          lock_b.acquire();
6.          lock_a.acquire();
7.          b = sum;
8.          a = x;
9.          lock_b.release();
10.         lock_a.release();
11.         cond = run;
        }
    }
    Tombo() {
1.      while (true) {
2.          lock_a.acquire()
3.          a = 0;
4.          lock_a.release()
5.          lock_b.acquire()
6.          b = 1;
7.          lock_b.release()
        }    }
```

Thread 1 runs `Kiki`, Thread 2 runs `Jiji`, and Thread 3 runs `Tombo`. Assuming round robin scheduling (threads are initially scheduled in numerical order):

1. What are the values of `a` and `b` after 50 CPU cycles?

2. What are the values of `a` and `b` after 200 CPU cycles? What line is the program counter on for each thread?

Now assume we use the same round robin scheduler but we just run 2 instances of `Kiki`.

3. What are the values of `a` and `b` after 100 cycles?

Now we replace our scheduler with a CFS-like scheduler. In particular, this scheduler is invoked on every call to `lock_acquire` or `lock_release`. We still have 2 threads running `Kiki`, but this time thread 1 runs for 50 cycles before thread 2 starts.

4. What are the values of `a` and `b` at the end of the 73rd cycle?