Section 6: Spin Locks, Scheduling and Fairness

September 26, 2015

Contents

1 Warmup

Which of the following are true about Round Robin Scheduling?

1. The average wait time is less that that of FCFS for the same workload.
2. Is supported by `thread_tick` in Pintos.
3. It requires pre-emption to maintain uniform quanta.
4. If quanta is constantly updated to become the # of cpu ticks since boot, Round Robin becomes FIFO.
5. If all threads in the system have the same priority, Priority Schedulers **must** behave like round robin.
6. Cache performance is likely to improve relative to FCFS.
7. If no new threads are entering the system all threads will get a chance to run in the cpu every `QUANTA*SECONDS_PER_TICK*NUMTHREADS` seconds. (Assuming `QUANTA` is in ticks).
8. This is the default scheduler in Pintos
9. It is the fairest scheduler

2 Vocabulary

- **Scheduler -** The process scheduler is a part of the operating system that decides which process runs at a certain point in time. It usually has the ability to pause a running process, move it to the back of the running queue and start a new process;
- **Spin Locks -** A type of lock where the implementation of **lock.acquire()** is to simply check if the lock is available in a loop (”spin”). Since the thread remains active but is not performing a useful task, the use of such a lock is a kind of busy waiting.

3 Problems

3.1 test_and_set

Assume that I use test_and_set to emulate the behavior of locks.

```
int value = 0;
int hello = 0;

void print_hello() {
    while (test_and_set(value));
    hello += 1;
    printf("Child thread: %d\n", hello);
    value = 0;
    pthread_exit(0);
}

void main() {
    pthread_t thread1;
    pthread_t thread2;
    pthread_create(&thread1, NULL, (void *) &print_hello, NULL);
    pthread_create(&thread2, NULL, (void *) &print_hello, NULL);
    while (test_and_set(value));
    printf("Parent thread: %d\n", hello);
    value = 0;
}
```

Assume the following sequence of events:
1. Main starts running and creates both threads and is then context switched right after
2. Thread2 is scheduled and run until after it increments hello and is context switched
3. Thread1 runs until it is context switched
4. The thread running main resumes and runs until it get context switched
5. Thread2 runs to completion
6. The thread running main runs to completion (but doesn't exit yet)
7. Thread1 runs to completion

Is this sequence of events possible? Why or why not?

And each step, if test_and_set is called, what value(s) will it return?

Given this sequence of events, what will C print?

Is this a good way to implement locks? Why or why not?

3.2 test_and_test_and_set?

To lower the overhead a more elaborate locking protocol test and test-and-set can be used. The main idea is not to spin in test-and-set but increase the likelihood of successful test-and-set by spinning until the lock seems like it is free.
Fill in the rest of the implementation for a test_and_test_and_set based lock:

```
int locked = 0;

void lock() {

  _______________________ // Spin until lock looks empty
  while (test_and_set(locked));
}

void unlock() {

  _______________________
}
```

Is this a better implementation of a lock that just using test_and_set? Why or why not?

Given the following implementations of lock and unlock, what are all the possible outputs that C might print out? Assume the pid for the main thread is 1 and the pid for created thread is 2. Also assume the print statements here are atomic.

```
int locked = 0;

void lock(pid_t pid) {
  while (locked == 1);
  printf("%s\n", "The lock is free!");
  while (test_and_set(locked));
  printf("Lock acquired by: %d\n", pid);
}

void unlock(pid_t pid) {
  locked = 0;
  printf("Lock released by: %d\n", pid);
}

void do_nothing(void* arg) {
  pid_t pid = getpid();
  lock(pid);
  unlock(pid);
  pthread_exit(0);
}

void main() {
  pthread_t thread;
  pthread_create(&thread, NULL, (void *) &do_nothing, NULL);
  do_nothing();
```

3.3 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);
}
```

3.3.1 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.

3.3.2 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?

3.3.3 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)

3.4 Totally Fair Scheduler

You design a new scheduler, you call it TFS. The idea is relatively simple, in the begining, we have three values `BIG_QUANTA`, `MIN_LATENCY` and `MIN_QUANTA`. We want to try and schedule all threads every `MIN_LATENCY` ticks, so they can get atleast a little work done, but we also want to make sure they run *atleast* `MIN_QUANTA` ticks. In addition to this we want to account for priorities. We want a threads priority to be inversely proportial to its `vruntime` or the amount of ticks its spent in the CPU in the last `BIG_QUANTA` ticks.

You may make the following assumptions in this problem:

- Priority scheduling in Pintos is functioning properly,
- Priority donation is not implemented.
- Alarm is not implemented.
- `thread_set_priority` is never called by the thread
- You may ignore the limited set of priorities enforced by pintos (priority values may span any `float` value)
- For simplicity assume floating point operations work in the kernel

3.4.1 Per thread quanta

How long will a particular thread run? (use the threads priority value)

3.4.2 struct thread

Below is the declaration of `struct thread`. What field(s) would we need to add to make TFS possible? You may not need all the blanks.

```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved stack pointer. */
    float priority;                       /* Priority, as a float. */
    struct list_elem allelem;           /* List element for all threads list. */

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /* List element. */

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif

    ___________________________         /* What goes here? */

    ___________________________         /* What goes here? */

    ___________________________         /* What goes here? */

    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
  };
```

3.4.3 thread tick

What is needed for `thread_tick()` for TFS to work properly? You may not need all the blanks.

```
void
thread_tick (void)
{
  struct thread *t = thread_current ();

  /* Update statistics. */
  if (t == idle_thread)
    idle_ticks++;
#ifdef USERPROG
  else if (t->pagedir != NULL)
    user_ticks++;
#endif
  else
    kernel_ticks++;

  ______________________________________________;

  ______________________________________________;
  /* Enforce preemption. */
  if (++thread_ticks >= TIME_SLICE) {  /* TIME_SLICE may need to be replaced with something else */
    intr_yield_on_return ();

    _________________________________;

    _________________________________;

    _________________________________;

    _________________________________;

    _________________________________;

    _________________________________;

    _________________________________;
  }
}
```

3.4.4 timer interrupt

What is needed for `timer_interrupt` for TFS to function properly.

```
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  _____________________________________;

  _____________________________________;
```

```
  -------------------------------------;

  -------------------------------------;

  -------------------------------------;

  -------------------------------------;

  -------------------------------------;

  -------------------------------------;

  -------------------------------------;

  -------------------------------------;

  -------------------------------------;

  -------------------------------------;

  thread_tick ();
}
```

3.4.5 thread create

What is needed for `thread_create()` for TFS to work properly? You may not need all the blanks.

```
tid_t
thread_create (const char *name, int priority,
               thread_func *function, void *aux)
{
  /* Body of thread_create omitted for brevity */

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

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

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

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

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

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

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

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

  ------------------------------------------------
```

```
  ------------------------------------------------

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

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

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

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

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

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


  /* Add to run queue. */
  thread_unblock (t);
  if (priority > thread_get_priority ())
    thread_yield ();

  return tid;
}
```

3.4.6 Analysis

Explain the high level behavior of this scheduler; what exactly is it trying to do? How is it different/similar from/to the multilevel feedback scheduler from the project?