# Section 7: Scheduling and Fairness

October 6, 2016

## 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

2,3,4,8 1. Easy to find counter example. 2. True. 3. True. 4. True. 5. False. Not a requirement. 6. False. More context switches means worse cache performance. 7. Trick question. There is some overhead. 8. True. 9. Trick question. Needs definition of fair.

# 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;

# 3 Problems

## 3.1 Locking and Conventions

In section 5, you may remember encountering race conditions inside of the Central Galactic Floopy Corporation's currency exchange server, which runs on top of pthreads. We said that we could make the transactions run correctly by making the balance increment/decrement atomic. We can make the increment/decrement pair appear atomic by adding a lock to each account, and acquiring the locks when we run the transaction.

```
typedef struct account_t {
  pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZED;
  int balance;
  long uuid;
};

void transfer(account_t *donor, account_t *recipient, float amount) {
  assert (donor != recipient);

  // lock accounts so we can make the transfer safely
  pthread_mutex_lock(&donor->lock);
  pthread_mutex_lock(&recipient->lock);

  // check balances and make transfer if possible
  if (donor->balance < amount) {
    printf("Insufficient funds.\n");
  } else {
    donor->balance -= amount;
    recipient->balance += amount;
  }

  // unlock accounts
  pthread_mutex_unlock(&recipient->lock);
  pthread_mutex_unlock(&donor->lock);
}
```

If we use the locking code given above, will our code run correctly? Have we introduced a new bug into our code? Can you give an example of where this code would fail?

The locking scheme we use above will occasionally deadlock. For example, if we tried to transfer money from Bob to Alice and from Alice to Bob in separate transactions at the same time, transaction 1 will acquire Bob's lock and wait on Alice's lock, while transaction 2 acquires Alice's lock while waiting on Bob's lock.

Can you modify the code above to resolve this bug?

```
typedef struct account_t {
  pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZED;
  int balance;
  long uuid;
};

void transfer(account_t *donor, account_t *recipient, float amount) {
  assert (donor != recipient);


  // lock accounts so we can make the transfer safely
  if (donor->uuid < recipient->uuid) {
    pthread_mutex_lock(&donor->lock);
    pthread_mutex_lock(&recipient->lock);
  } else {
    pthread_mutex_lock(&recipient->lock);
    pthread_mutex_lock(&donor->lock);
  }


  // check balances and make transfer if possible
  if (donor->balance < amount) {
    printf("Insufficient funds.\n");
  } else {
    donor->balance -= amount;
    recipient->balance += amount;
  }

  // unlock accounts
  pthread_mutex_unlock(&recipient->lock);
  pthread_mutex_unlock(&donor->lock);
}
```

## 3.2 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. It 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 only ever be 0 or 1
- The priority of the thread can be accessed in the field `int priority` in `struct thread`
- When the scheduler needs to pick a new thread to run, it will select the head of the ready queue

- Don't 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);
  if (t->priority == 1) {
    list_push_front (&ready_list, &t->elem);
  else
    list_push_back (&ready_list, &t->elem);
  t->status = THREAD_READY;
  intr_set_level (old_level);
}
}
```

### 3.2.1 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.

**Downgrade priority when thread uses up its quanta, upgrade priority when it voluntarily yields, or gets blocked.**

### 3.2.2 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?

**The insert operations are cheaper, and it provides a good approximation to priority scheduling.**

### 3.2.3 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)

**We can have more than 2 priorities but still a small number of fixed priorities, and have a queue for each priority, and then pop off threads from each queue as necessary.**

### 3.3 Life Ain't Fair

Suppose the following threads denoted by THREADNAME : PRIORITY pairs arrive in the ready queue at the clock ticks shown. Assume all threads arrive unblocked and that each takes 5 clock ticks to finish executing. Assume threads arrive in the queue at the beginning of the time slices shown and are ready to be scheduled in that same clock tick. (This means you update the ready queue with the arrival before you schedule/execute that clock tick.) Assume you only have one physical CPU.

```
0   Devin : 7
1
2   Josh : 1
3   Andrew: 3
4
5   Aleks : 5
6
7   Thurston: 11
8
9   Cory: 14
```

Determine the order and time allocations of execution for the following scheduler scenarios:

- Shortest Time Remaining First (SRTF/SJF) WITH preemptions
- Preemptive priority (higher is more important)
- Round Robin with time slice 3

Write answers in the form of vertical columns with one name per row, each denoting one clock tick of execution. For example, allowing Devin 3 units at first looks like:

```
0   Devin
1   Devin
2   Devin
```

Assume that threads that arrive always get scheduled earlier than threads that have already been running or have just finished.

It will probably help you to draw a diagram of the ready queue at each tick for this problem.

Explanation for RR:

From t=0 to t=3, Devin gets to run since there is initially no one else on the run queue. At t=3, Devin gets preempted since the time slice is 3. Josh is selected as the next person to run, and Andrew gets added to the run queue (t=2.9999999) just before Devin (t=3).

Josh is the next person to run from t=3 to t=6. At t=5, Aleks gets added to the run queue, which consists of at this point: Andrew, Devin, Aleks

At t=6, Josh gets preempted and Andrew gets to run since he is next. Josh gets added to the back of the queue, which consists of: Devin, Aleks, Josh.

From t=6 to t=9, Andrew gets to run and then is preempted. Devin gets to run again from t=9 to t=10, and then finishes executing. Aleks gets to run next and this pattern continues until everyone has completed running.

```
RR:
0   Devin
1   Devin
2   Devin
3   Josh
```

```
4   Josh
5   Josh
6   Andrew
7   Andrew
8   Andrew
9   Devin
10  Devin
11  Aleks
12  Aleks
13  Aleks
14  Josh
15  Josh
16  Thurston
17  Thurston
18  Thurston
19  Cory
20  Cory
21  Cory
22  Andrew
23  Andrew
24  Aleks
25  Aleks
26  Thurston
27  Thurston
28  Cory
29  Cory

Preemptive SRTF
0   Devin
1   Devin
2   Devin
3   Devin
4   Devin
5   Josh
6   Josh
7   Josh
8   Josh
9   Josh
...
(Pretty much just like FIFO since every thread takes 5 ticks)

Preemptive Priority
0   Devin
1   Devin
2   Devin
3   Devin
4   Devin
5   Aleks
6   Aleks
7   Thurston
8   Thurston
9   Cory
10  Cory
11  Cory
```

```
12  Cory
13  Cory
14  Thurston
15  Thurston
16  Thurston
17  Aleks
18  Aleks
19  Aleks
20  Andrew
21  Andrew
22  Andrew
23  Andrew
24  Andrew
25 - 29 Josh
```

## 3.4 [PRIORITY: 0] Totally Fair Scheduler

*You probably won't be able to allocate any time to this question in section. If not, please make sure you work through this, and check the solution when it's posted.*

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 proportional 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)

Every thread $T_k$ will run for
$max(\frac{T_k.priority}{\sum_{i=0}^{n} T_i.priority} \cdot$ `MIN_LATENCY` , `MIN_QUANTA` )

### 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

    int vruntime;
    int quanta;
    /* 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++;

  t->vruntime++;
  /* Enforce preemption. */
  if (++thread_ticks >= t->quanta){
    intr_yield_on_return ();
    t->priority =  (1.0/t->vruntime);
    float total_priority = 0.0f;
    for (e = list_begin (&all_list); e != list_end (&all_list);
         e = list_next(e)) {
      struct thread *t = list_entry (e, struct thread, allelem);
      total_priority += t->priority;
    }
    t->quanta = max(t->priority/total_priority*MIN_LATENCY, MIN_QUANTA);
  }
}
```

### 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++;
  if (ticks % BIG_QUANTA == 0) {
    int tc = list_size(all_list);

    for (e = list_begin (&all_list); e != list_end (&all_list);
         e = list_next (e)) {
      struct thread *t = list_entry (e, struct thread, allelem);
      t->vruntime = 0;
      t->priority = 1.0f;
      t->quanta = max((1.0/tc)*MIN_LATENCY, MIN_QUANTA);
    }
```

```
  }
  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 */
  old_level = intr_disable ();
  int total_priority = 0;
  for (e = list_begin (&all_list); e != list_end (&all_list);
       e = list_next (e)) {
      struct thread *t = list_entry (e, struct thread, allelem);
      total_priority += t->priority;
  }

  for (e = list_begin (&all_list); e != list_end (&all_list);
       e = list_next (e)) {
    struct thread *t = list_entry (e, struct thread, allelem);
    t->quanta = max((t->priority/total_priority)*(MIN_LATENCY), MIN_QUANTA);
  }
  intr_set_level (old_level);
  /* 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?

This scheduler is a "fair" scheduler it tries to treat the CPU as a shared "ideal" CPU that multiplexes fairly between all the processes weighted by their priorities. Both this and multilevel feedback are similar in that they try and give threads who've used the CPU recently lower priority; they are dissimilar in the fact that the per tick operations are faster. And the MFSQ has more memory: it remembers about its CPU time for a longer period of time (well, it's slightly more complicated).