Section 4: Scheduling and Synchronization

CS162

July 8, 2019

Contents

1 Warmup **2**

2 Vocabulary **2**

3 Problems **4**

3.1 Locking Up the Floopies . . . 4
3.2 Scheduling . . . 4
3.3 Simple Priority Scheduler . . . 5
3.3.1 Fairness . . . 6
3.3.2 Better than Priority Scheduler? . . . 7
3.3.3 Tradeoff . . . 7
3.4 Totally Fair Scheduler . . . 7
3.4.1 Per thread quanta . . . 7
3.4.2 struct thread . . . 8
3.4.3 thread tick . . . 8
3.4.4 timer interrupt . . . 9
3.4.5 thread create . . . 10
3.4.6 Analysis . . . 11
3.5 test_and_set . . . 11
3.6 Hello World . . . 12
3.7 SpaceX Problems . . . 13

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;
- **FIFO Scheduling** - First-In-First-Out (aka First-Come-First-Serve) scheduling runs jobs as they arrive. Turnaround time can degrade if short jobs get stuck behind long ones (convoy effect);
- **Round-Robin Scheduling** - Round-Robin scheduling runs each job in fixed-length time slices (quanta). The scheduler preempts a job that exceeds its quantum and moves on, cycling through the jobs. It avoids starvation and is good for short jobs, but context switching overhead can become important depending on quanta length;
- **Priority Scheduling** - Priority scheduling runs the highest priority job, based on some assigned priorities. Starvation of low-priority jobs and priority inversion (a higher priority task waiting for a lower priority one, usually for a lock) are issues. Priorities can be static or dynamic, and if dynamic can change based on heuristics or locking-related donations;
- **SRTF Scheduling** - Shortest Remaining Time First scheduling runs the job with the least remaining amount of computation time and is preemptive. It has the optimally shortest average turnaround time. In practice remaining computation time can't be predicted, so SRTF is often used as a post-facto benchmark for other algorithms;
- **Multi-Level Feedback Queue Scheduling** - MLFQS uses multiple queues with priorities, dropping CPU-bound jobs that consume their entire quanta into lower-priority queues;
- **Condition Variable** - A synchronization variable that provides serialization (ensuring that events occur in a certain order). A condition variable is associated with:
  - a lock (a condition variable + its lock are known together as a **monitor**)

  - some boolean condition (e.g. `hello < 1`)
  - a queue of threads waiting for the condition to be true

  In order to access any CV functions **OR** to change the truthfulness of the condition, a thread must/should hold the lock. Condition variables offer the following methods:

  - **cv_wait(cv, lock)** - Atomically unlocks the lock, adds the current thread to **cv**'s thread queue, and puts this thread to sleep.
  - **cv_notify(cv)** - Removes one thread from **cv**'s queue, and puts it in the ready state.
  - **cv_broadcast(cv)** - Removes all threads from **cv**'s queue, and puts them all in the ready state.

  When a **wait()**ing thread is notified and put back in the ready state, it also re-acquires the lock before the **wait()** function returns.
  When a thread runs code that may potentially make the condition true, it should acquire the lock, modify the condition however it needs to, call notify() or broadcast() on the condition's CV, so waiting threads can be notified, and finally release the lock.
  Why do we need a lock anyway? Well, consider a race condition where thread 1 evaluates the condition C as false, then thread 2 makes condition C true and calls **cv.notify**, then 1 calls **cv.wait** and goes to sleep. Thread 1 might never wake up, since it went to sleep too late.

- **Hoare Semantics** - (In terms of condition variable) Wake a blocked thread when the condition is true and transfer control of the CPU and ownership of the lock to that thread immediately. This is difficult to implement in practice and generally not used despite being conceptually easier to deal with.

- **Mesa Semantics** - (In terms of condition variable) Wake a blocked thread when the condition is true, with no guarantee that the thread will execute immediately. The newly woken thread simply gets put on the ready queue and is subject to the same scheduling mechanisms as any other thread. The implication of this is that **you must check the condition with a while loop instead of an if statement because it is possible for the condition to change to false between the time the thread was unblocked and the time it takes over the CPU**.

```
test_and_set(int * value) {
    int result = *value;
    *value = 1;
    return result;
}
```

↑ does this atomically

- expensive op

3 Problems

3.1 Locking Up the Floopies

In section 3, 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. The Central Galactic Floopy Corporation hires a consultant named Morty who suggests making the increment/decrement pair appear atomic by adding a lock to each account, and acquiring the locks when we run the transaction.

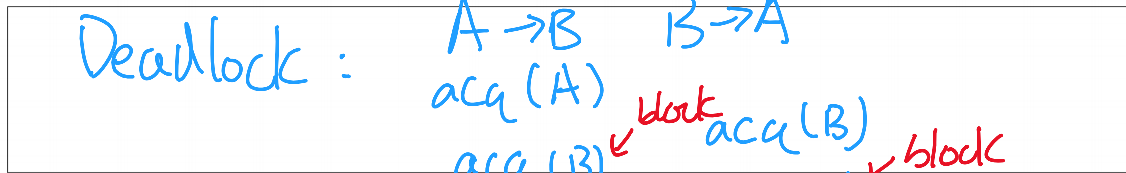
```
typedef struct account_t {
  pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZED;
  int balance;
  long uuid;
};

void transfer(account_t *donor, account_t *recipient, float amount) {
  // 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 the code run correctly? Has Morty introduced a new bug into our code? Can you give an example of where this code would fail?



Can you modify the code above to resolve this bug?

3.2 Scheduling

Consider the following single-threaded processes and their arrival times, CPU bursts, and priority

Process	Arrival Time	CPU Burst	Priority
A	1	5	1
B	3	3	3
C	5	2	2
D	4	4	4

Please note:

- Priority scheduler is preemptive.
- Newly arrived processes are scheduled last for RR. When the RR quanta expires, the currently running thread is added at the end of to the ready list before any newly arriving threads.
- Break ties via priority in Shortest Remaining Time First (SRTF).
- If a process arrives at time x, they are ready to run at the beginning of time x.
- Ignore context switching overhead.
- The quanta for RR is 1 unit of time.
- Total turnaround time is the time a process takes to complete after it arrives.

Fill in the following scheduling table and calculate the total turnaround time for each scheduling algorithm

B higher priority than A, same time remaining

Time	FIFO	RR	Queue	SRTF	Priority
1 A:5,1	A	A	A	A	A
2	A	A	A	A	A
3 B:3,3	A	A	AB	B	B
4 D:4,4	A	B	BAD	B	D
5 C:2,2	A 5	A	ADBC	B 3	D
6	B	D	DBCA	C	D
7	B	B	BCAD	C 3	D 4
8	B 6	C	CADB	A	B
9	D	A 9	ADBC	A	B 7
10	D	D	DBC	A 10	C
11	D	B 9	BCD	D	C 7
12	D 9	C 8	CD	D	A
13	C	D	D	D	A
14	C 10	D 11		D 11	A 14
Total Turnaround Time	30	37		27	32

Σ (optimal)

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);

  _______________________________________________ See soln

  _______________________________________________

  _______________________________________________

  _______________________________________________
  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.

priority ↓ when it uses up quanta
priority ↑ if thread voluntarily yields ("friendly thread")
↑ good for user input (waits for user input and yields, but need to respond to action with high 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?

less bins (not as much granularity), but operations are much faster: O(1) inserts

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)

less bins than normal but more than just 2.

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:

SE Solns for 3.4

- 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?

3.5 test_and_set

In the following code, we use test_and_set to emulate 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;
}
```

← T1 gets stuck

0 = free
1 = "locked"

← M gets stuck

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

"locked"

T&S
1. N/A
2. 0
3. 1 (repeated)
4. 1 (repeated)
5. N/A
6. 0
7. 0

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

M and T1 block until T2 resets value to 0.

At each step where `test_and_set(&value)` is called, what value(s) does it return?

See above

Given this sequence of events, what will C print?

C : 1
P : 1
C : 2

Is this implementation better than using locks? Explain your rationale.

No, busy waiting. Like our default sleep method in Pintos.

3.6 Hello World

This code compiles (given a sprinkling of `#includes` etc.) but doesn't work properly. Why?

```
pthread_mutex_t lock;
pthread_cond_t cv;
int hello = 0;

void* print_hello(void* arg) {
    hello += 1;
    printf("First line (hello=%d)\n", hello);
    pthread_cond_signal(&cv);
    pthread_exit(0);
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, print_hello, NULL);
    while (hello < 1) {
      pthread_cond_wait(&cv, &lock);
```

also should initialize lock

needed to have the lock acquired before calling wait

```
    }
    printf("Second line (hello=%d)\n", hello);
    return 0;
}
```

Add in the necessary code to the above problem to make it work correctly.

3.7 SpaceX Problems

Consider this program.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;

int n = 3;

void* counter(void* arg) {
  pthread_mutex_lock(&lock);
  for (n = 3; n > 0; n--)
    printf("%d\n", n);
  pthread_cond_signal(&cv);
  pthread_mutex_unlock(&lock);
}

void* announcer(void* arg) {
  while (n != 0) {
    pthread_mutex_lock(&lock);
    pthread_cond_wait(&cv, &lock);
    pthread_mutex_unlock(&lock);
  }
  printf("FALCON HEAVY TOUCH DOWN!\n");
}

int main() {
  pthread_t t1, t2;
  pthread_create(&t1, NULL, counter, NULL);
  pthread_create(&t2, NULL, announcer, NULL);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  return 0;
}
```

idea: we want ownership of n until wait call

we read n without any synchro !!

move lock outside of while loop

ex. 1) announcer enters while loop when n != 0, blocks on lock
2) counter signals & unlocks lock
3) announcer waits but never gets signalled.

What is wrong with this code?