Section 5: Synchronization and Scheduling

September 19, 2015

Contents

1 Warmup

1.1 Hello World

Will this code compile/run?
Why or why not?

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

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

void main() {
    pthread_t thread;
    pthread_create(&thread, NULL, (void *) &print_hello, NULL);
    while (hello < 1) {
      pthread_cond_wait(&cv, &lock);
    }
    printf("Second line (hello=%d)\n", hello);
}
```

This won't work because the main thread should have locked the lock before calling pthread_cond_wait, and the child thread should have locked the lock before calling pthread_cond_signal. (Also, we never initialized the lock and cv.)

2 Vocabulary

- **Lock** - Synchronization variables that provide mutual exclusion. Threads may acquire or release a lock. Only one thread may hold a lock at a time. If a thread attempts to acquire a lock that is held by some other thread, it will block at that line of code until the lock is released and it successfully acquires it. Implementations can vary.
- **Scheduler** - Routine in the kernel that picks which thread to run next given a vacant CPU and a ready queue of unblocked threads. See next_thread_to_run() in Pintos.
- **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.

- **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**.

3 Problems

3.1 Hello Word Continued

Add in the necessary code to the warmup to make it work correctly.

Acquire a lock before the cv is used and release it afterwards.

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

void main() {
    pthread_t thread;
    pthread_mutex_init(&lock, 0);
    pthread_cond_init(&cv, 0);

    pthread_create(&thread, NULL, (void *) &print_hello, NULL);

    pthread_mutex_lock(&lock);
    while (hello < 1) {
        pthread_cond_wait(&cv, &lock);
    }
    pthread_mutex_unlock(&lock);

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

3.2 Spot the Problem

What is wrong with this code?

```
pthread_mutex_t lock;
pthread_cond_t cv;
int n = 3;
void counter() {
  pthread_mutex_lock(&lock);
  for (n = 3; n > 0; n--)
    printf("%d\n", n);
  pthread_cond_signal(&cv);
  pthread_mutex_unlock(&lock);
}
void announcer() {
  while (n != 0) {
    pthread_mutex_lock(&lock);
    pthread_cond_wait(&cv, &lock);
    pthread_mutex_unlock(&lock);
  }
  printf("BLAST OFF!\n");
}
```

The lock in announcer() should be outside of the while loop. Or else, the announcer thread might never wake up.

3.3 Baking with Condition Variables

A number of people are trying to bake cakes. Unfortunately, they each know only one skill, so they need to all work together to bake cakes. Use independent threads (one person is one thread) which communicate through condition variables to solve the problem. A skeleton has been provided, fill in the blanks to make the implementation work.

A cake requires:

- 1 cake batter
- 2 eggs

Instructions:

1. Add ingredients to bowl
2. Heat bowl (it's oven-safe)
3. Eat the cake, clean out the bowl, and go back to step 1

Requirements:

- Don't start heating the cake in the oven unless there are exactly the right number of ingredients in the bowl.
- Don't add raw ingredients to a currently-baking cake or a finished cake.
- Don't eat the cake unless it's done baking.
- Given enough time, the code should bake an unbounded number of cakes, and should never stop.

```
int numBatterInBowl = 0;
int numEggInBowl = 0;
bool readyToEat = false;
pthread_mutex_t lock;
pthread_cond_t needIngredients;
pthread_cond_t readyToBake;
pthread_cond_t startEating;

void batterAdder()
{
  pthread_mutex_lock(&lock);
  while (1) {

    ---------------------------------------------- {

      ------------------------------
    }
    addBatter(); // Sets numBatterInBowl += 1

    ------------------------------
  }
}

void eggBreaker()
{
  pthread_mutex_lock(&lock);
  while (1) {

    ---------------------------------------------- {

      ------------------------------
    }
    addEgg(); // Sets numEggInBowl += 1

    ------------------------------
  }
}

void bowlHeater()
{
  pthread_mutex_lock(&lock);
  while (1) {

    ------------------------------------------------------------ {

      ------------------------------
    }
    heatBowl(); // Sets readyToEat = true, numBatterInBowl = 0, numEggInBowl = 0

    ------------------------------
  }
}

void cakeEater()
{
  pthread_mutex_lock(&lock);
  while (1) {

    ---------------------------------------------- {

      ------------------------------
    }
    eatCake(); // Sets readyToEat = false and cleans the bowl for another cake

    ------------------------------
  }
}

int main(int argc,char *argv[])
{
  // Initialize mutex and condition variables
  // Start threads: 1 batterAdder, 2 eggBreakers, 1 bowlHeater, and 1 cakeEater
  // main() sleeps forever
```

```
  int numBatterInBowl = 0;
  int numEggInBowl = 0;
  bool readyToEat = false;
  pthread_mutex_t lock;
  pthread_cond_t needIngredients;
  pthread_cond_t readyToBake;
  pthread_cond_t startEating;
```

```
void batterAdder()
{
  pthread_mutex_lock(&lock);
  while (1) {
    while (numBatterInBowl || readyToEat) {
      pthread_cond_wait(&needIngredients, &lock);
    }
    addBatter(); // Sets numBatterInBowl += 1
    pthread_cond_signal(&readyToBake);
  }
}

void eggBreaker()
{
  pthread_mutex_lock(&lock);
  while (1) {
    while (numEggInBowl >= 2 || readyToEat) {
      pthread_cond_wait(&needIngredients, &lock);
    }
    addEgg(); // Sets numEggInBowl += 1
    pthread_cond_signal(&readyToBake);
  }
}

void bowlHeater()
{
  pthread_mutex_lock(&lock);
  while (1) {
    while (numBatterInBowl != 1 || numEggInBowl != 2) {
      pthread_cond_wait(&readyToBake, &lock);
    }
    heatBowl(); // Sets readyToEat = true, numBatterInBowl = 0, numEggInBowl = 0
    pthread_cond_signal(&startEating);
  }
}

void cakeEater()
{
  pthread_mutex_lock(&lock);
  while (1) {
    while (!readyToEat) {
      pthread_cond_wait(&startEating, &lock);
    }
    eatCake(); // Sets readyToEat = false and cleans the bowl for another cake
    pthread_cond_broadcast(&needIngredients);
  }
}

int main(int argc,char *argv[])
```

```
{
  // Initialize mutex and condition variables
  // Start threads: 1 batterAdder, 2 eggBreakers, 1 bowlHeater, and 1 cakeEater
  // main() sleeps forever
}
```

3.4 These Are The Locks You're Looking For

What does C print in the following code? You may not assume anything about the scheduler other than that it behaves with Mesa semantics. (i.e. could be RR, FIFO, priority) In general, user programs should not depend on the scheduler and should run correctly regardless of the scheduler used.

```
int ben = 0;

void main() {
    pthread_t thread;
    pthread_create(&thread, NULL, &helper, NULL);
    pthread_yield();
    if (ben==1) printf("These are not the droids you are looking for. ben = %d\n", ben);
    else printf("These are the droids you are looking for! ben = %d\n", ben);
    exit(0);
}

void *helper(void* arg) {
    ben+=1;
    pthread_exit(0);
}
```

```
The output of this program is undefined because there is a race condition
on the global variable ben.
The value of ben could change while it is being read by main,
or it could change after the value is checked but before the print is
executed.
Yields have no control over scheduling.
```

Declare a lock and use it to guarantee the print message of this program. Pseudocode is fine.

```
int ben = 0;
//LOCK L

void main() {
    pthread_t thread;
    //ACQUIRE L
    pthread_create(&thread, NULL, &helper, NULL);
    pthread_yield();
    if (ben==1) printf("These are not the droids you are looking for.\n");
    else printf("These are the droids you are looking for!\n");
    //RELEASE L
    exit(0);
}

void *helper(void* arg) {
    //ACQUIRE L
    ben+=1;
    //RELEASE L
    pthread_exit(0);
}

First you must declare the lock as a global variable (so it goes in the data segment
and can be accessed by all threads.) Then with appropriate locking, the best you can
do in this case is making it always print "These are the droids you are looking for!"

Locks do not guarantee ordering, only mutual exclusion. If we acquire the lock
after the new thread is created, we guarantee that there is no race condition
on the variable 'ben' but not its actual value, as the scheduler could preempt
the main thread and run the new
thread even before a yield is called.
```

Suppose we did the following instead to attempt to force serialization. What will this program print? Does it add any extra synchronization protection?

```
int ben = 0;

void main() {
    //INTR_DISABLE()
    pthread_t thread;
    pthread_create(&thread, NULL, &helper, NULL);
    pthread_yield();
    if (ben==1) printf("These are not the droids you are looking for.\n");
    else printf("These are the droids you are looking for!\n");
    //INTR_ENABLE()
    exit(0);
}

void *helper(void* arg) {
    ben+=1;
    pthread_exit(0);
}
```

```
We still don't know for sure because disabling interrupts does not prevent you
from forcing your current thread to yield. (Note that yielding isn't an interrupt.)

If the scheduler schedules the new thread after the yield,
it will always print "These are not the droids you are looking for."
If the scheduler schedules the main thread after the yield,
it will always print "These are the droids you are looking for!"

Because interrupts are disabled, the race condition on ben is gone
but like with locks the order of execution is still undefined.
```

3.5 Only A Sith Deals In Absolute Conditions

Consider the same block of code. How do you ensure that you always print out the canonically correct line? Assume the scheduler behaves with Mesa semantics. (Pseudocode is OK) You may only add lines, so the trivial answer of not checking the value of ben before printing is not correct.

```
int ben = 0;

void main() {
    pthread_t thread;
    pthread_create(&thread, NULL, &helper, NULL);
    pthread_yield();
    if (ben==1) printf("These are not the droids you are looking for.\n");
    else printf("These are the droids you are looking for!\n");
    exit(0);
}

void *helper(void* arg) {
    ben+=1;
    pthread_exit(0);
}
```

```
int ben = 0;
//LOCK = L
//CONDVAR = C

void main() {
    pthread_t thread;
    //LOCK L ACQUIRE
    pthread_create(&thread, NULL, &helper, NULL);
    pthread_yield();
    //WHILE BEN != 1
        //CONDVAR C WAIT
    if (ben==1) printf("These are not the droids you are looking for.\n");
    //SHOULD ALWAYS BE TRUE
    else printf("These are the droids you are looking for!);
    //LOCK L RELEASE
    exit(0);
}

void *helper(void* arg) {
    //LOCK L ACQUIRE
    ben+=1;
    //CONDVAR C SIGNAL
    //LOCK L RELEASE
    pthread_exit(0);
}

(Did not bother to use syntax for POSIX locks and condvars since it isn't in PintOS)
```

3.6 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   Roger : 7
1
2   Jackson : 1
3   Andrew: 3
4
5   Aleks : 5
6
7   Will: 11
8
9   Alec: 14
```

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

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

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

```
0   Roger
1   Roger
2   Roger
```

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

We're assuming that threads that arrive always get scheduled earlier than threads that have already been running or have just finished.

Explanation for RR:

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

Jackson 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, Roger, Aleks

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

From t=6 to t=9, Andrew gets to run and then is preempted. Roger 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   Roger
1   Roger
2   Roger
3   Jackson
4   Jackson
```

```
5   Jackson
6   Andrew
7   Andrew
8   Andrew
9   Roger
10  Roger
11  Aleks
12  Aleks
13  Aleks
14  Jackson
15  Jackson
16  Will
17  Will
18  Will
19  Alec
20  Alec
21  Alec
22  Andrew
23  Andrew
24  Aleks
25  Aleks
26  Will
27  Will
28  Alec
29  Alec

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

Preemptive Priority
0   Roger
1   Roger
2   Roger
3   Roger
4   Roger
5   Aleks
6   Aleks
7   Will
8   Will
9   Alec
10  Alec
11  Alec
12  Alec
13  Alec
```

```
14  Will
15  Will
16  Will
17  Aleks
18  Aleks
19  Aleks
20  Andrew
21  Andrew
22  Andrew
23  Andrew
24  Andrew
25 - 29 Jackson
```

3.7 All Threads Must Die

You have three threads with the associated priorities shown below. They each run the functions with their respective names. Assume upon execution all threads are initially unblocked and begin at the top of their code blocks. The operating system runs with a preemptive priority scheduler. You may assume that set_priority commands are atomic.

Tyrion : 4
Ned: 5
Gandalf: 11

Note: The following uses references to Pintos locks and data structures.

```
struct list braceYourself;    // pintos list. Assume it's already initialized and populated.
struct lock midTerm;          // pintos lock. Already initialized.
struct lock isComing;

void tyrion(){
    thread_set_priority(12);
    lock_acquire(&midTerm);
    lock_release(&midTerm);
    thread_exit();

}

void ned(){
    lock_acquire(&midTerm);
    lock_acquire(&isComing);
    list_remove(list_head(braceYourself));
    lock_release(&midTerm);
    lock_release(&isComing);
    thread_exit();
}

void gandalf(){
    lock_acquire(&isComing);
    thread_set_priority(3);
    while (thread_get_priority() < 11) {
        printf("YOU .. SHALL NOT .. PAAASS!!!!!!);
        timer_sleep(20);
    }
    lock_release(&isComing);
    thread_exit();
}
```

What is the output of this program when there is no priority donation? Trace the program execution and number the lines in the order in which they are executed.

```
void tyrion(){
5    thread_set_priority(12);
6    lock_acquire(&midTerm); //blocks
```

```
    lock_release(&midTerm);
    thread_exit();

}

void ned(){
3    lock_acquire(&midTerm);
4    lock_acquire(&isComing);  //blocks
    list_remove(list_head(braceYourself));
    lock_release(&midTerm);
    lock_release(&isComing);
    thread_exit();
}

void gandalf(){
1    lock_acquire(&isComing);
2    thread_set_priority(3);
7    while (thread_get_priority() < 11) {
8        printf("YOU .. SHALL NOT .. PAAASS!!!!!!); //repeat till infinity
9        timer_sleep(20);
    }
    lock_release(&isComing);
    thread_exit();
}

Gandalf, as you might expect, endlessly prints "YOU SHALL NOT PASS!!" every 20 clock ticks or so.
```

What is the output and order of line execution if priority donation was implemented? Draw a diagram of the three threads and two locks that shows how you would use data structures and struct members (variables and pointers, etc) to implement priority donation for this example.

```
void tyrion(){
8    thread_set_priority(12);
9    lock_acquire(&midTerm); //blocks
    lock_release(&midTerm);
    thread_exit();

}

void ned(){
3    lock_acquire(&midTerm);
4    lock_acquire(&isComing);  //blocks
12   list_remove(list_head(braceYourself));  //KERNEL PANIC
    lock_release(&midTerm);
    lock_release(&isComing);
    thread_exit();
}

void gandalf(){
1    lock_acquire(&isComing);
2    thread_set_priority(3);
5    while (thread_get_priority() < 11) {   //priority is 5 first, but 12 at some later loop
6        printf("YOU .. SHALL NOT .. PAAASS!!!!!!);
7        timer_sleep(20);
    }
10    lock_release(&isComing);
11    thread_exit();
}
```

It turns out that Gandalf generally does mean well. Donations will make Gandalf allow you to pass.
At some point Gandalf will sleep on a timer and leave Tyrion alone in the ready queue.
Tyrion will run even though he has a lower priority (Gandalf has a 5 donated to him)
Tyrion then sets his priority to 12 and chain-donates to Gandalf. Gandalf breaks his loop.
Ned unblocks after Gandalf exits.
However, allowing Ned to remove the head of a list will trigger an ASSERT failure in lib/kernel/list.c.

Gandalf will print YOU SHALL NOT PASS at least once.
Then Ned will get beheaded and cause a kernel panic that crashes Pintos.