Section 6

Tuesday, May 21, 2019 8:35 AM

section6

Section 6: Synchronization and Scheduling

October 2-3, 2018

Contents

1 Warmup **2**
1.1 Hello World 2

2 Vocabulary **2**

3 Problems **3**
3.1 Hello World Continued 3
3.2 SpaceX Problems 4
3.3 I Love You, Unconditionally! 5
3.4 Locking Up the Floopies 6
3.5 All Threads Must Die 8
3.6 Baking with Condition Variables 10

1 Warmup

1.1 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);
    }
    printf("Second line (hello=%d)\n", hello);
    return 0;
}
```

= PTHREAD_MUTEX_INITIALIZER
= pthread_cond_init

Lock.acquire();

Lock.release;

The child thread can update hello before the main thread perofrms its first check on hello. It can bypass the while loop, write second line, before the first thread rpints the "first" line

2 Vocabulary

MX synch var.

- **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.

seems li
Not explic
The spirit

- **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.

speak w

ke a scenario, though previously
itly defined as bad, does not follow
of the priority system.

ith the hand of the king!

- **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 World Continued

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

MX

Serialization

3.2 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;
}
```

Macro

What, incredible, POWER!

Buggy interleaving case 1:
Counter gets the lock
Announcer finds n != 0 and waits for the lock
Counter decrements n to 0. It sends a signal to NOBODY.
It then release the lock.
Announcer gets the lock, then yields it to wait on the conditior
It waits forever!

What is wrong with this code?

3	Get the lock before checking for the event.
2	If the event happens while you're waiting ofr the lcok, you'll
1	Begin waiting later, and miss the event :(
---	

n variable.

3.3 I Love You, Unconditionally!

Consider the following program:

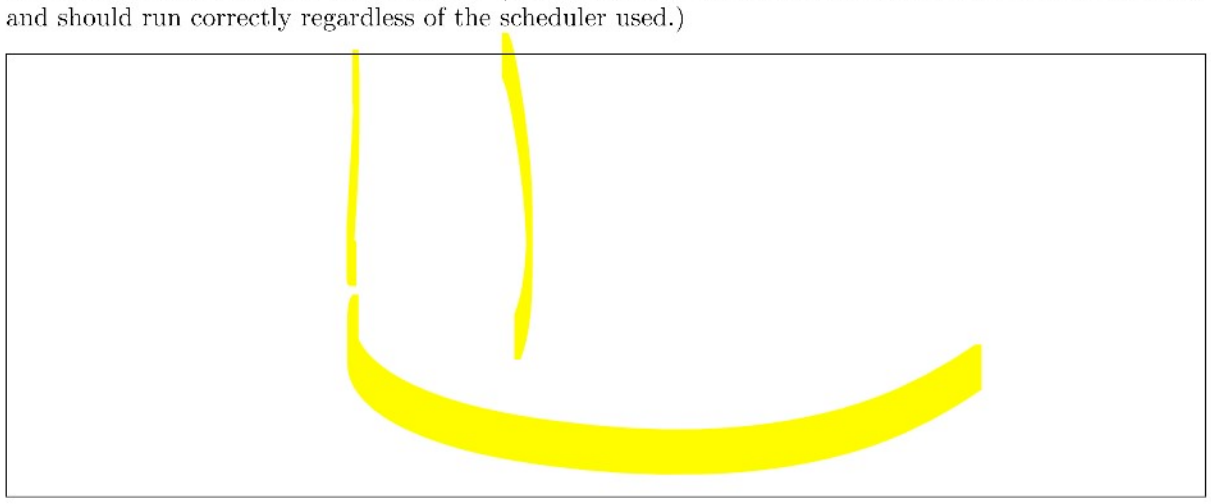
```
int subaru = 0;

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, &helper, NULL);
    pthread_yield();
    if (subaru==1) printf("I love Emilia!\n");
    else printf("I love Rem!\n");
    return 0;
}

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

pthread_join(thread, NULL)

Modify the program above such that we always print out the canonically correct line ("I love Emilia!", unfortunately). **You may only add lines.** You may not assume anything about the scheduler other than that it behaves with Mesa semantics. (In general, user programs should not depend on the scheduler and should run correctly regardless of the scheduler used.)



3.4 Locking Up the Floopies

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. 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?

When it runs, it will affect the correct behavior.
However, there is a non-zero probability the systemw ill enter deadlock, never terminate,
And therefore fail at affecting it's task.

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) {
// lock accounts so we can make the transfer safely

  _If (&donor->uuid < &recipeint->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);
}
```

3.5 All Threads Must Die

You have three Pintos threads with the associated priorities shown below. They each run the functions with their respective names.

Tyrion : 4
Ned: 5
Gandalf: 11

Assume upon execution that all threads are 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. (Note: The following uses references to Pintos locks and data structures.)

```
struct list brace_yourself;   // pintos list. Assume it's already initialized and populated.
struct lock midterm;          // pintos lock. Already initialized.
struct lock is_coming;

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

void ned(){
    lock_acquire(&midterm);
    lock_acquire(&is_coming);
    list_remove(list_head(brace_yourself));
    lock_release(&midterm);
    lock_release(&is_coming);
    thread_exit();
}

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

Only the one true king, w/ priority >= 12
Will move me from my weird pattern!

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.

Scheduler schedules gandalf.
He gets the is_coming lock ,then sets his priorty to 3
This preempts Ned to run, who gets the midterm lock but blocks on is_coming
Tyrion is next to run, sets his priority to 12, stays running, but blocks on midterm lock.
Gandalf again comes off the scheduler, and repeatedly saysy ous hall not pass, until the end of time

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.

3.6 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.

(Hint: start with the cake eater; we've filled out a bit more of his duties.)

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

```
void* eggBreaker(void* arg)
{
  pthread_mutex_lock(&lock);
  while (1) {
    while (numEggsInBowl == 2 || readyToEat) {
      Pthread_cond_wait(&needIngredients, &lock);
    }
    addEgg(); // Sets numEggInBowl += 1
    Pthread_cond_signal(&readyToBake);
  }
}

void* bowlHeater(void* arg)
{
  pthread_mutex_lock(&lock);
  while (1) {
    while (!(numEggsInBowl == 2 && numBatterInBowl == 1)) {
      Pthread_cond_wait(&readyToBake, &lock);
    }
    heatBowl(); // Sets readyToEat = true, numBatterInBowl = 0, numEggInBowl = 0
    Pthread_cond_signal(&readyToEat);
  }
}

void* cakeEater(void* arg)
{
  pthread_mutex_lock(&lock);
  while (1) {
    while (!readyToEat) {
      Pthread_cond_wait(&start_eating, &lock);
    }
    eatCake(); // Sets readyToEat = false and cleans the bowl for another cake
    Pthread_cond_broadcast(&need_ingredients, &lock);
  }
}

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
}
```