Section 5: Synchronization, Condition Variables

CS 162

July 8, 2020

Contents

1 Vocabulary

- **Condition Variable** - A synchronization variable that provides serialization (ensuring that events occur in a certain order). A condition variable is defined by:
  - 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 a 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 a condition variable, wake a blocked thread when the condition is true with no guarantee on when that thread will actually execute. (The newly woken thread simply gets put on the ready queue and is subject to the same scheduling semantics as any other thread.) The implications of this mean 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.

2 Synchronization

2.1 Bounded Buffer with Locks and Semaphores

In class we discussed a solution to the bounded-buffer problem for multiple producers and multiple consumers using semaphores and locks. In this problem, there is a fixed size buffer. Producers add to the buffer, but only if there is room in the buffer. Consumers remove from the buffer, but only if there are items. Producers should sleep until there is space in the buffer and consumers should sleep until there is something in the buffer.

1. Reproduce this solution from lecture. Think about how we can use semaphores track how many empty and full spots are currently in the buffer. Do not go onto the next part until you are done with this, as it will give away the answer.

```
Producer () {                    Consumer () {
  emptyBuffers.P()                 fullBuffers.P()
  mutex.Acquire()                  mutex.Acquire()
  queue.Enqueue()                  queue.Dequeue()
  mutex.Release()                  mutex.Release()
  fullBuffers.V()                  emptyBuffers.V()
}                                }
```

2. Examine the following implementation where the lock operations have been moved. Explain it is correct. If not, explain why.

```
Producer () {                    Consumer () {
  mutex.Acquire()                  mutex.Acquire()
  emptyBuffers.P()                 fullBuffers.P()
  queue.Enqueue()                  queue.Dequeue()
  fullBuffers.V()                  emptyBuffers.V()
  mutex.Release()                  mutex.Release()
}                                }
```

This code is incorrect. It can lead to deadlock. What is happening here is the producers and consumers can go to sleep while holding the lock, preventing others from even touching the queue. Consider the case where the buffer is full. Suppose a Producer comes and grabs the mutex and then waits for emptyBuffers. A consumer then hangs on mutex and no one will ever dequeue to empty the buffer.

3. Examine the following implementation where the lock operations have been moved. Explain it is correct. If not, explain why.

```
Producer () {                    Consumer () {
  mutex.Acquire()                  fullBuffers.P()
  emptyBuffers.P()                 mutex.Acquire()
  queue.Enqueue()                  queue.Dequeue()
  fullBuffers.V()                  mutex.Release()
  mutex.Release()                  emptyBuffers.V()
}                                }
```

This code is incorrect. It can lead to deadlock. The problem is exactly as the first case of deadlock mentioned in part a. The procedure can wait on the queue and sleep with the lock, but the consumer will never get to emptyBuffers.V() since that requires having held the lock. The lesson here is that a thread should never sleep while holding a shared lock.

4. Examine the following implementation where the lock operations have been moved. Explain it is correct. If not, explain why.

```
Producer () {                    Consumer () {
  emptyBuffers.P()                 fullBuffers.P()
  mutex.Acquire()                  mutex.Acquire()
  queue.Enqueue()                  queue.Dequeue()
  fullBuffers.V()                  emptyBuffers.V()
  mutex.Release()                  mutex.Release()
}                                }
```

This code is correct. It *may* have reduced concurrency, as you incur the possibility of waking up a consumer with fullBuffers.V() only for it to sleep on the lock.

2.2 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.)

2.3 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;
}
```

What is wrong with this code?

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

2.4 Condition Variables

Consider the following block of code. How do you ensure that you always print out "Yeet Haw"? 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("Yeet Haw\n");
    } else {
        printf("Yee Howdy\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("Yeet Haw\n");
    } else { ... }
    //LOCK L RELEASE
    exit(0);
}

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

2.5 CS162 (Online) Office Hours

Suppose we want to use condition variables to control access to a CS162 (digital) office hours room for three types of people: students, TA's, and professors. A person can attempt to enter the room (or will wait outside until their condition is met), and after entering the room they can then exit the room. The follow are each type's conditions:

- Suppose professors get easily distracted and so they need solitude, with no other students, TA's, or professors in the room, in order to enter the room.
- TA's don't care about students inside and will wait if there is a professor inside, but there can only be up to 3 TA's inside (any more would clearly be imposters from CS161 or CS186).
- Students don't care about other students of TA's in the room, but will wait if there is a professor inside.

To summarize the constraints:

- Professor must wait if anyone else is in the room
- TA must wait if there are already 3 TA's in the room
- TA must wait if there is a professor in the room
- student must wait if there is a professor in the room

```
typedef struct lock { . . . } lock        // lock.acquire(),lock.release()
typedef struct cv { . . . } cv            // cv.wait(&lock),cv.signal(), cv.broadcast()

#define TA_LIMIT 3
typedef struct {
    lock lock;
    cv student_cv;
    int waitingStudents, activeStudents;
    cv ta_cv, prof_cv;
    int waitingTas, waitingProfs;
    int activeTas, activeProfs;
} room_lock;
```

```
/* mode = 0 for student, 1 for TA, 2 for professor */
enter_room(room_lock *rlock, int mode) {
    rlock->lock.acquire();
    if (mode == 0) {
        while ((rlock->activeProfs+rlock->waitingProfs) > 0) {
            rlock->waitingStudents++;
            rlock->student_cv.wait(&rlock->lock);
            rlock->waitingStudents--;
        }
        rlock->activeStudents++;
    } else if (mode == 1) {
        while((rlock->activeProfs+rlock->waitingProfs) > 0 || rlock->activeTas >= TA_LIMIT) {
            rlock->waitingTas++;
            rlock->ta_cv.wait(&rlock->lock);
            rlock->waitingTas--;
        }
```

```
        rlock->activeTas++;
    } else {
        while((rlock->activeProfs + rlock->activeTas + rlock->activeStudents) > 0) {
            rlock->waitingProfs++;
            rlock->prof_cv.wait(&rlock->lock);
            rlock->waitingProfs--;
        }
        rlock->activeProfs++;
    }
    rlock->lock.release();
}

exit_room(room_lock *rlock, int mode) {
    rlock->lock.acquire();
    if (mode == 0) {
        rlock->activeStudents--;
        if ((rlock->activeStudents + rlock->activeTas) == 0 && rlock->waitingProfs) {
                rlock->prof_cv.signal();
        }
    } else if (mode == 1) {
        rlock->activeTas--;
        if ((rlock->activeStudents + rlock->activeTas) == 0 && rlock->waitingProfs) {
            rlock->prof_cv.signal();
        } else if (rlock->activeTas < TA_LIMIT && rlock->waitingTas) {
            rlock->ta_cv.signal();
        }
    } else {
        rlock->activeProfs--;
        if (rlock->waitingProfs) {
            rlock->prof_cv.signal();
        } else {
            if (rlock->waitingTas)
                rlock->ta_cv.broadcast();
            if (rlock->waitingStudents)
                rlock->student_cv.broadcast();
        }
    }
    rlock->lock.release();
}
```

3 Project Design

This problem is designed to help you with implementing wait and argument passing in your project. Recall that wait suspends execution of the parent process until the child process specified by the parameter id exits, upon which it returns the exit code of the child process. In Pintos, there is a 1:1 mapping between processes and threads.

3.1 Brainstorm

"wait" requires communication between a process and its children, usually implemented through shared data. The shared data might be added to struct thread, but many solutions separate it into a separate structure. At least the following must be shared between a parent and each of its children:

- Child's exit status, so that "wait" can return it.
- Child's thread id, for "wait" to compare against its argument.
- A way for the parent to block until the child dies (usually a semaphore).
- A way for the parent and child to tell whether the other is already dead, in a race-free fashion (to ensure that their shared data can be freed).

3.2 Code

Data structures to add to thread.h for waiting logic:

```
Pseudocode:

process_wait (tid_t child_tid) {
    iterate through list of child processes
    if child process tid matches tid parameter, call sema_down
    on the semaphore associated with that child process (when
    child process exits, it will sema_up on that same semaphore,
    waking up the parent process)
    --- after waking ---
    set exit code to terminated child process's exit code
    decrease ref_cnt of child //why? need to free memory, "zombie processes"
    return exit_code
}

process_exit (void) {
    sema_up on semaphore that parent might be sleeping on
    remove all child processes from child process list
    decrement ref_cnt
}


Code:

struct wait_status
    {
      struct list_elem elem;      /* 'children' list element. */
      struct lock lock;           /* Protects ref_cnt. */
      int ref_cnt;                /* 2=child and parent both alive,
             1=either child or parent alive,
```

```
            0=child and parent both dead. */
    tid_t tid;                  /* Child thread id. */
    int exit_code;              /* Child exit code, if dead. */
    struct semaphore dead;      /* 1=child alive, 0=child dead. */
  };


struct wait_status *wait_status; /* This process's completion state. */
struct list children;            /* Completion status of children. */
```

Implement wait:

```
    - Find the child in the list of shared data structures.
      (If none is found, return -1.)

    - Wait for the child to die, by downing a semaphore in the
      shared data.

    - Obtain the child's exit code from the shared data.

    - Destroy the shared data structure and remove it from the
      list.

    - Return the exit code.
```

Implement exit:

```
    - Save the exit code in the shared data.

    - Up the semaphore in the data shared with our parent
      process (if any).  In some kind of race-free way (such
      as using a lock and a reference count or pair of boolean
      variables in the shared data area), mark the shared data
      as unused by us and free it if the parent is also dead.

    - Iterate the list of children and, as in the previous
      step, mark them as no longer used by us and free them if
      the child is also dead.

      - Terminate the thread.
```

3.3 Argument Passing

Fill out the following code to copy integer arguments onto the stack. You can change the ESP by modifying `if_->esp`.

```
void populate_stack_args(int argc, uint32_t *argv, struct intr_frame *if_) {
    // The stack pointer is available in if_->esp (which has the type uint32_t*)
    int i;
```

```
    for (_____________________________________) {

        if_->esp = ____________________;

        _________________ = argv[i];
    }
}
```

```
void populate_stack_args(int argc, uint32_t* argv, struct intr_frame* if_) {
    // The stack pointer is available in if_->esp (which has the type uint32_t*)
    int i;

    for (i = argc-1; i >= 0; i--) {

        if_->esp = if_->esp - 4;

        *if_->esp = argv[i];
    }
}
```