CS 537: Intro to Operating Systems (Fall 2017)

Worksheet 8 - Locks and CVs

Due: Nov 1st 2017 (Wed) in-class OR email Simmi before 11:59 pm

1. Locks

a. Assume we have a new instruction called the **LoadAndStoreZero (LASZ)**, and it does the following atomically (here is C pseudo-code):

```
int LoadAndStoreZero(int *addr) {
    int old = *addr;
    *addr = 0;
    return old;
}
```

Build a working spin lock using the new instruction **LoadAndStoreZero (LASZ)**.

NOTE: For full points, your code should adhere to the C syntax.

```
typedef struct __lock_t {


} lock_t;


void init(lock_t *lock) {


}

void acquire(lock_t *lock) {






}

void release(lock_t *lock) {



}
```

b. Consider the following implementation for a lock in Solaris using **Queues, Test-And-Set, Yield, and Wakeup**.

```
typedef struct __lock_t {
    int         flag;
    int         guard;
    queue_t     *q;
} lock_t;

void lock_init(lock_t *lock) {
    lock->flag = lock->guard = 0;
    lock->q    = queue_init();
}

void lock(lock_t *lock) {
    while (xchg(&lock->guard, 1) == 1)
        ; // spin
    if (lock->flag == 0) {
        lock->flag = 1;
        lock->guard = 0;
    } else {
        queue_push(lock->q, gettid());
        setpark();
        lock->guard = 0;
        park();
    }
}

void unlock(lock_t *lock) {
    while (xchg(&lock->guard, 1) == 1)
        ; // spin
    if (queue_empty(lock->q))
        lock->flag = 0;
    else
        unpark(queue_pop(lock->q));
    lock->guard = 0;
}
```

The following are the definitions of the key routines:

- **park()**: Puts the calling thread to sleep.
- **unpark(threadID)**: Wake up the thread with the given threadID.
- **setpark()**: A thread indicates that it's about to park.

i. What is the **purpose** of the **guard lock**? What may happen if we don't have the **guard lock**? **Explain** with a simple **example**.

ii. Assume thread 1 is currently holding the **flag lock** and is inside the **critical section**. During this time, there are **9 more threads** that are currently waiting in the queue for the lock to be released. You may assume that there are no more threads waiting for this lock. You may also assume that all these threads will acquire the flag lock only once. Under this scenario, how many times will the flag lock be **released** by these 10 threads? **Explain** the reasoning behind your **answer**.

iii. What will happen if **park()** is called **before releasing the guard lock**?

iv. This lock still **spins** while trying to acquire the **guard** lock. So, is this better in any way than simple spin locks with respect to **performance**? Explain your answer.

2. Condition Variables

a. Assume the following implementation for the famous **producer/consumer** problem.

```
void put(int value) {
  buffer[fillptr] = value;
  fillptr = (fillptr + 1) % max;
  count++;
}
```

```
int get() {
  int tmp = buffer[useptr];
  useptr = (useptr + 1) % max;
  count--;
  return tmp;
}
```

```
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    P_mutex_lock(&mutex);          //p1
    if (count == max)              //p2
      P_cond_wait(&empty,&mutex);//p3
    put(i);                        //p4
    P_cond_signal(&fill);          //p5
    P_mutex_unlock(&mutex);        //p6
  }
}
```

```
void *consumer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    P_mutex_lock(&mutex);        //c1
    if (count == 0)              //c2
      P_cond_wait(&fill,&mutex);//c3
    int tmp = get();             //c4
    P_cond_signal(&empty);       //c5
    P_mutex_unlock(&mutex);      //c6
    printf("%d\n", tmp);
  }
}
```

Assume further that the only way a thread stops running is when it explicitly blocks in either a condition variable or lock (in other words, no untimely interrupts switch from one thread to the other).Also assume there are NO SPURIOUS WAKEUPs from wait().

i. In the following, show which lines of code (from p1 - p6 and c1 - c6) run given a particular scenario. Scenario 0 is completed for you as an example.
Scenario 1: 1 producer (P), 1 consumer (C), max = 1. Producer P runs first. Stop when consumer C has consumed one entry.

```
P: p1, p2, p4, p5, p6, p1, p2, p3
C:                                c1, c2, c4
```

Scenario 2: 1 producer (P), 1 consumer (C), max = 1. Consumer C runs first. Stop when consumer C has consumed one entry.

```
P:

C:
```

Scenario 3: 1 producer (P), 2 consumers (Ca, Cb), max = 1. Consumer Ca runs first, then P, then Cb. Stop when each consumer has consumed one entry

P:

Ca:

Cb:

ii. Are there any **bugs** in this implementation? If so, how do you **fix** them?

b. Here is some code for the producer/consumer problem we were trying to solve in class today. We'll use a new primitive: **Pthread_cond_broadcast()**. Unlike traditional signaling, this wakes up ALL threads waiting on a condition. Here is some code using such a broadcast:

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == MAX)
            Pthread_cond_wait(&cv, &mutex);
        put(i);
        Pthread_cond_broadcast(&cv); // here!
        Pthread_mutex_unlock(&mutex);
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 0)
            Pthread_cond_wait(&cv, &mutex);
        int tmp = get();
        Pthread_cond_broadcast(&cv); // here!
        Pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

Does this code work? If **yes**, then explain **why** does this code work? If **no**, then explain **why not**?