Locks

Simple Spin Lock

Prove that the lock given below violates one or more of the neccessary lock properties. Use sequential consistency.

```
volatile int lock = 0;

void lock() {
  while(lock != 0) {/*wait here*/}
  lock = 1;
}

void unlock() { lock = 0; }
```

Give an alternative for a two thread lock. Prove (using sequential consistency) that it provides mutual exclusion and deadlock freedom.

Simple Spin Lock Solution

We are testing if this lock provides mutual exclusion. If not, both processes will be in the critical region at the same time. For this the following sequences of reads and writes must happen:

$A : R(lock, 0) \rightarrow W(lock, 1) \rightarrow$ Thread A is in CR now
$B : R(lock, 0) \rightarrow W(lock, 1) \rightarrow$ Thread B is in CR now

It is easy to see that it is possible to build a sequentially consistent interleaving of those operations:

$R_A(lock, 0) \rightarrow R_B(lock, 0) \rightarrow W_A(lock, 1) \rightarrow W_B(lock, 1) \rightarrow$ Thread A and B are in CR now.

The Peterson lock (given below) provides mutual exclusion and deadlock freedom (tid is the thread id, which can be 0 or 1, so each thread can identify the other thread as 1-tid).

```
flag[tid] = 1;
victim = tid;
while (flag[1-tid] && (victim == tid)) {};
// CR
flag[tid] = 0;
```

Mutual Exclusion

We are testing if this lock provides mutual exclusion. If not, both processes will be in the critical region at the same time. For this the following sequences of reads and writes must happen:

$0 : W(flag[0], 1) \rightarrow W(victim, 0) \rightarrow R(flag[1]) \rightarrow R(victim) \rightarrow CR_0$ (1)
$1 : W(flag[1], 1) \rightarrow W(victim, 1) \rightarrow R(flag[0]) \rightarrow R(victim) \rightarrow CR_1$ (2)

Assume that thread 0 was the last thread to write into victim:

$W_1(victim, 1) \rightarrow W_0(victim, 0)$ (3)

This implies that thread 0 reads victim as 0. Therefore, to enter the critical section, it must have read flag[1] as 0.

$W_0(victim, 0) \rightarrow R_0(flag[1], 0)$ (4)

If we combine the equations 2-4 we get

$W_1(flag[1], 1) \rightarrow W_1(victim, 1) \rightarrow W_0(victim, 0) \rightarrow R_0(flag[1], 0)$

This is a contradiction, as $want[1]$ is set to 1 and later observed to be 0, with no other writes in between.

Deadlock Free

Suppose the given lock is not deadlock free, both processes are spinning in the while loop. For that to happen flag[0] and flag[1] must both be one. Victim must always be zero for process zero, and one for process one. Since victim is not written to by any of the processes, this is impossible, as it can not have two different values simultaneously.

Filter Lock

Prove that the filter lock (given below) provides mutual exclusion for n threads. Use sequential consistency.

```
volatile int level[n] = {0,0,...,0};
volatile int victim[n];

void lock() {
  for (int l=1; l<n; l++) {
    level[tid] = l;
    victim[l] = tid;
    while ((∃k != tid) (level[k] >= l && victim[l] == tid)) {};
  }
}

void unlock() { level[tid] = 0; }
```

Mutual Exclusion

We will proof that for $0 \leq j \leq n$ there are at most n-j threads at level j. We do that by induction.

For j=0, the base case, this is trivially true.

For the induction step the induction hypothesis implies that there are at most n-j+1 threads at level j-1. To show that at least one thread cannot progress to level j, we argue by contradiction: assume there are n-j+1 threads at level j. Let A be the last thread at level j to write to victim[j]. Because A is last, for any other B at level j:

$W_B(victim[j]) \rightarrow W_A(victim[j])$

From the code we see that B writes level[B] before it writes victim[j]

$W_B(level[B] = j) \rightarrow W_B(victim[j]) \rightarrow W_A(victim[j])$

also from the code we see that A reads level[B] after writing to victim[j]

$W_B(level[B] = j) \rightarrow W_B(victim[j]) \rightarrow W_A(victim[j]) \rightarrow R_A(level[B])$

Because B is at level j, every time A reads level[B], it observes a value greater than or equal to j, implying that A could not have completed the waiting loop, a contradiction.

Deadlock Freedom

We argue by reverse induction on the levels. The base case, level n - 1, is trivial, because it contains at the most one thread. For the induction hypothesis, assume that every thread that reaches level j + 1 or higher, eventually enters (and leaves) its critical section.

Suppose A is stuck at level j. Eventually, by the induction hypothesis, there are no threads at higher levels. Once A sets level[A] to j, then any thread at level j - 1 that subsequently reads level[A] is prevented from entering level j. Eventually, no more threads enter level j from lower levels. All threads stuck at level j are in the waiting loop, and the values of the victim and level fields no longer change.

We now argue by induction on the number of threads stuck at level j. For the base case, if A is the only thread at level j or higher, then clearly it will enter level j + 1. For the induction hypothesis, we assume that fewer than k threads cannot be stuck at level j. Suppose threads A and B are stuck at level j. A is stuck as long as it reads victim[j] = A, and B is stuck as long as it reads victim[j] = B. The victim field is unchanging, and it cannot be equal to both A and B, so one of the two threads will enter level j + 1, reducing the number of stuck threads to k - 1, contradicting the induction hypothesis.

Bakery Lock

Prove that the bakery lock (given below) provides mutual exclusion for n threads. Use sequential consistency.

```
volatile int flag[n] = {0, 0, ..., 0};
volatile int label[n] = {0, 0, ...,0};
void lock() {
  flag[tid] = 1;
  label[tid] = max(label[0], ..., label[n-1]) + 1;
  while ((∃k != tid)(flag[k] && (label[k],k) ≪ (label[tid],tid))) {};
}

public void unlock() { flag[tid] = 0; }
```

Note: (a,b) $\ll$ (c,d) behaves like a $<$ c, unless a=c, in which case it becomes b $<$ d.

Suppose the algorithm does not provide mutual exclusion. Let A and B be two threads concurrently in the critical section. Let $labeling_A$ and $labeling_B$ be the last respective sequences of acquiring new labels prior to entering the critical section (line 5). Suppose that (label[A], A) $\ll$ (label[B], B). When B successfully completed the test in its waiting section, it must have read that flag[A] was false or that (label[B], B) $\ll$ (label[A], A). However, for a given thread, its tid is fixed and its label[] values are strictly increasing, so B must have seen that flag[A] was false. It follows that

$labeling_B \rightarrow R_B(flag[A]) \rightarrow W_A(flag[A]) \rightarrow labeling_A$

which contradicts the assumption that (label[A], A) $\ll$ (label[B], B).