CS 347M Spring 2019: Quiz 2 (Total: 20 Marks)

Name: SARTHAK CONSUL **Roll number:** 16D100012

1. **[5 marks]** Consider the following synchronization problem. A group of children are picking chocolates from a box that can hold up to N chocolates. A child that wants to eat a chocolate picks one from the box to eat, unless the box is empty. If a child finds the box to be empty, she wakes up the mother, and waits until the mother refills the box with N chocolates. Unsynchronized code snippets for the child and mother threads are as shown below:

```
//Child
while True:
  getChocolateFromBox()
  eat()

//Mother
while True:
  refillChocolateBox(N)
```

You must now modify the code of the mother and child threads by adding suitable synchronization such that a child invokes `getChocolateFromBox()` only if the box is non-empty, and the mother invokes `refillChocolateBox(N)` only if the box is fully empty. Solve this question using only locks and condition variables, and no other synchronization primitive. The following variables have been declared for use in your solution; your solution must use only these variables and no others for synchronization.

```
int count = 0;  //the number of chocolates present in the box
mutex m; // you may invoke lock and unlock
condvar fullBox, emptyBox; //you may perform wait and signal
```

(a) Code for child thread

```
lock(m);
while(count==0){
   signal(emptybox);
   wait(fullbox, m);
}getChocolateFromBox();
count--;   //child east 1 chocolate

unlock(m);
```

Initially count=0, child sleeps mother sees that and calls 'wait' in the beginning, causing deadlock

Missed 'eat()' but still it is okay

(b) Code for mother thread

```
lock(m);
while(count==0){
  wait(emptybox,m);
  refillbox(N);
  count = N;
  signal(fullbox);
unlock(m);
```

2. **[5 marks]** Repeat the above question, but your solution now must use only semaphores and no other synchronization primitive. The following variables have been declared for use in your solution.

```
int count = 0;  //the number of chocolates present in the box
semaphore m = 1, fullBox = 0, emptyBox = 0;
//initial values of semaphores are as specified above
//you may invoke up and down methods on a semaphore
```

Further, you are constrained to use the semaphores `m`, `fullBox`, and `emptyBox` as binary semaphores. That is, the count of the semaphore variable is restricted to be 0 or 1 only.

(a) Code for child thread

```
down(emptybox);
down(m);
if(count==0){
  up(emptybox);
  down(fullbox);
}
else{
  getChocolateFromBox;
  count--;
  up(m);
}
```

(b) Code for mother thread

```
down(emptybox);
down(m);
count = N;
refillbox(N);
up(m);
up(fullbox);
```

```
up(emptybox);
if(count==0){
  refillbox(N);
  up(fullbox);
  down(m)
  count = N;
  up(m);
}
```

3. **[3 marks]** Consider a multi-threaded program, where threads need to acquire and hold multiple locks at a time. To avoid deadlocks, all threads are mandated to use the function `acquire_locks`, instead of acquiring locks independently. This function takes as arguments a variable sized array of pointers to locks (i.e., addresses of the lock structure), and the number of lock pointers in the array, as shown in the function prototype below. That is, any thread that wishes to acquire one or more locks will create an array, store pointers to the locks it wishes to acquire in this array, and invoke the `acquire_locks` functions. The function returns once all locks have been successfully acquired.

```
void acquire_locks(struct lock *la[], int n);
//i-th lock in array can be locked by calling lock(la[i])
```

Describe (in English, or in pseudocode) one way in which you would implement this function, while ensuring that no deadlocks happen during lock acquisition. Your solution must not use any other locks beyond those provided as input. Note that multiple threads can invoke this function concurrently, possibly with an overlapping set of locks, and the lock pointers can be stored in the array in any arbitrary order. You may assume that the locks in the array are unique, and there are no duplicates within the input array of locks. You are not allowed to make assumptions about any other fields (e.g., the presence of a unique identifier) in the lock structure itself. This is like the Philosopher's Dinner problem ~~as it~~ but with N utensils.

~~Same approach: have a~~ We cannot have a waiter (as no extra lock is permitted)

~~0~~ Have some ordering in acquiring locks.

2 ✓

4. **[2 marks]** Recall that the atomic instruction compare-and-swap (CAS) works as follows: `CAS(&var, oldval, newval)` writes `newval` into `var` and returns true if the old value of `var` is `oldval`. If the old value of `var` is not `oldval`, `CAS` returns false and does not change the value of the variable. Write code for the function to acquire a simple spinlock using the `CAS` instruction.

```
while( CAS(&lock, 1, 1) == 1) ;
```

0

5. **[3 marks]** The simple spinlock implementation studied in class does not guarantee any kind of fairness or FIFO order amongst the threads contending for the spin lock. A ticket lock is a spinlock implementation that guarantees a FIFO order of lock acquisition amongst the threads contending for the lock. Shown below is the code for the function to acquire a ticket lock. In this function, the variables `next_ticket` and `now_serving` are both global variables, shared across all threads, and initialized to 0. The variable `my_ticket` is a variable that is local to a particular thread, and is not shared across threads. The atomic instruction `fetch_and_increment(&var)` atomically adds 1 to the value of the variable and returns the old value of the variable.

```
acquire():
  my_ticket = fetch_and_increment(&next_ticket)
  while(now_serving != my_ticket); //busy wait
```

You are now required to write the code to release the spinlock, to be executed by the thread holding the lock. Your implementation of the release function must guarantee that the next contending thread (in FIFO order) will be able to acquire the lock correctly. You must not declare or use any other variables.

```
release(): //your code here
    if(my_ticket == now_serving){
        now_serving++;
    }
```

6. **[2 marks]** Consider a thread that has acquired and is holding a spinlock. For each question below, answer yes/no and provide a justification.

   (a) If the thread is holding a `pthread` spinlock in userspace, can this thread be safely preempted by the kernel to service an interrupt, without potentially causing a deadlock in the system? Yes, kernel can service an interrupt without causing deadlock if the ISR (i.e. function invoked by interrupt does not use the same lock. (which it won't as ISR is kernel code and the lock is user defined)

   (b) If the thread is holding a kernel spinlock in kernel space, can this thread be safely preempted to service an interrupt or for any other purpose, without potentially causing a deadlock in the system?

No.
When thread holding lock is interrupted, kernel

No.
If service routine invokes that spinlock(), it would remain spinning forever, as the lock is held by another thread and there is no way to context switch to that thread.