9 Monitors [15 points]

Recall that condition variables are used within monitors. Your task is to implement the functionality of condition variables using semaphores. Assume that you have a mutex implementation in a class called Mutex, and this class implements the Lock() and Unlock() methods. Similarly, assume that you have a semaphore implementation in a class called Semaphore, and this class implements the Down() and Up() methods.

Part A [12 points]

You are required to implement the Wait() and Signal() methods shown below. List any classes and variables you use and initialize them. If your implementation makes any assumptions, state them.

```
class Condition { int count = 0;   Semaphore sem = 0; };
Mutex lock;
Condition cond;

Condition::Wait(Mutex lock) {
    count++;
    lock.Unlock();
    sem.Down();
    lock.Lock();
    count--;
}

Condition::Signal(Mutex lock) {
    if (count > 0) {
        sem.Up();
    }
}
```

Part B [3 points]

Show an example function that invokes the Wait() method correctly.

```
function() {
    lock.Lock();
    cond.Wait(lock); // Wait needs to be called within a lock
    lock.Unlock();
}
```

10 Semaphores [15 points]

David, Sean, and Frank plant seeds continuously. David digs the holes. Sean then places a seed in each hole. Frank then fills the hole up. There are several synchronization constraints:

1. Sean cannot plant a seed unless at least one empty hole exists, but Sean does not care how far David gets ahead of Sean.
2. Frank cannot fill a hole unless at least one hole exists in which Sean has planted a seed, but the hole has not yet been filled. Frank does not care how far Sean gets ahead of Frank.
3. Frank does care that David does not get more than MAX holes ahead of Frank. Thus, if there are MAX unfilled holes, David has to wait.
4. There is only one shovel with which both David and Frank need to dig and fill the holes, respectively.

Write the pseudocode for the 3 processes which represent David, Sean and Frank using semaphores as the synchronization mechanism. Make sure to initialize the semaphores.

```
Semaphore shovel = 1;
Semaphore unfilled = MAX;
Semaphore seed = 0;
Semaphore fill = 0;
```

```
David()                Sean()                 Frank()
{                      {                      {
  while (1) {            while (1) {            while (1) {
    wait(shovel);          wait(seed);            wait(fill);
    wait(unfilled);        // seed                wait(shovel);
    // dig                 signal(fill);          // fill
    signal(seed);        }                        signal(unfilled);
    signal(shovel);    }                          signal(shovel);
  }                                             }
}                                             }
```

4. Synchronization [10 marks]

You have been hired to coordinate people trying to cross a river. There is a single boat, capable of holding at most three people. It will sink if more than three people board it at a time. The boat moves back and forth between the left and the right banks of the river. It is supposed to start moving only when it has three people in it. Otherwise, it waits for more people to arrive and board the boat.

You are asked to design the software to ensure than no failures occur when people use the boat to cross the river. For example, people shouldn't try to board a boat that is on the other bank. People can arrive on either bank. You model each person as a separate thread as follows:

```
Person(int location)
// location is either 0 or 1;
// 0 = left bank, 1 = right bank of the river
{
     BoardBoat(location); // write code for this function
     CrossRiver(location); // given to you
     GetOffBoat(location);  // write code for this function
}
```

`Boardboat(location)` must return only when the person has boarded the boat at `location` and the boat is about to start moving. At this point, `CrossRiver()` moves the boat to the other bank. `GetOffBoat()` is called to indicate that the person has stepped off the boat. Note that the `location` parameter to `GetOffBoat()` is the initial location (not destination location). Make sure that all people in the boat get off the boat before people going in the other direction are allowed in the boat. You need to write the `BoardBoat()` and `GetoffBoat()` functions below. Assume the `CrossRiver()` function is given to you. It returns when the boat gets to the other bank but does not depend on or affect your code in any other way. We have provided you some variables to help you. Fill the rest of the code as needed.

```
Monitor Boat {
    // assume both functions below acquire a lock on entry,
    // release the lock on exit
    int boat_direction = 0;// 0 when boat is going from
                           // left to right bank, otherwise
    int count = 0;         // nr of people in boat


    // add any condition variables needed
    // CV cv; Use syntax: cv.wait(), cv.signal()
     CV boat_available, boat_full;

    BoardBoat(int location) { // write < 12 lines of code
    // must return only when person has boarded boat and
    // boat is ready to move.

       while (boat_direction != location || count >= 3) {
           boat_available.wait();
       }
       count++;
       if(count < 3) {
           boat_full.wait();
       } else {
           boat_full.broadcast();
       }

    }

    GetOffBoat(int location){ // write < 6 lines of code
    // person is stepping off the boat
    // Don’t add a loop in the code.
    // hint: when is the boat ready to change direction

       --count;
       if(count == 0) {
           // we arrived on the other side, update boat direction
           boat_direction = !location;
           boat_available.broadcast();
       }

    }
}
```

Question 4. Synchronization Problem [10 MARKS]

In one form of passing messages between threads, called a *rendezvous*, both sending and receiving a message are blocking operations. That is, a thread calling the `msg_send()` function will block until the recipient calls `msg_recv()`. Similarly, if a thread calls `msg_recv()`, it must block until some other thread calls `msg_send()`. When sending a message, the intended destination must be specified, however, messages can be received from any source, with the id of the sender returned along with the message.

Assume that messages are of a fixed length, `MSG_LEN`, and that thread ids are chosen from a small range of integer values, `[0..TID_MAX-1]`. We define an array of *mailboxes*, one per thread, indexed by thread id, to help implement the functions. The definition of the mailbox structure is shown below.

```
struct mailbox {
        char *dest_buf; /* set by receiver */
        int sender_id;  /* set by sender */
        struct semaphore *recvr_ready; /* Initially, count = 0 */
        struct semaphore *sender_done; /* Initially, count = 0 */
};

struct mailbox mailboxes[TID_MAX];
```

Part (a) [8 MARKS] Complete the implementation of the message passing functions `msg_send()` and `msg_recv` on the following page. Remember that a thread may receive messages from multiple senders. *[You should at least read the partial implementation before answering the following two questions.]*

Part (b) [1 MARK] What is the main *advantage* of using blocking sends and receives in this way?

No intermediate storage is needed to hold sent messages until the receiver is ready to receive them.

Also acceptable: The sender knows the receiver has the message when it returns from send (or else it has an error immediately, although the functions on the following page don't return errors), which may simplify programming since we don't need to check separately that the receiver got the message.

Part (c) [1 MARK] What is the main *disadvantage* of using blocking sends and receives?

The sender must wait for the receiver before the message can be sent, instead of going on to other work. The receiver also waits, but this may be less of a problem if we assume the receiver needs to get the message before it can continue with its work anyway.

This question continues on the following page

```
/* Add the necessary calls to P() and V() on the mailbox semaphores.
 * You do not need to add code to every blank space.
 */
void msg_send(char *msg, int recvr_id) {
        struct mailbox *mbox = &mailboxes[recvr_id];

        /* SOLN: Wait for the receiver to indicate it is ready */

        P(mbox->recvr_ready);


        memcpy(mbox->dest_buf, msg, MSG_LEN);





        mbox->sender_id = get_current_thread_id();
        /* SOLN: Signal that the sender is finished sending the message */

        V(mbox->sender_done);


}
void msg_recv(char *msg, int *sender_id) {
        struct mailbox *mbox = &mailboxes[get_current_thread_id()];





        mbox->dest_buf = msg;
        /* SOLN: After setting dest_buf, signal ready to get msg */

        V(mbox->recvr_ready);

        /* SOLN: Wait for sender to indicate message is available */

        P(mbox->sender_done);
        *sender_id = mbox->sender_id;





}
```

4. Mutual Exclusion [8 marks]

Consider the following solution to the critical section problem for 2 threads:

```
1.      boolean blocked[2];
2.      int  turn;
3.      void P (int id) {
4.          while (true) {
5.              /* lines 6-12: Entry to Critical Section */
6.              blocked[id] = true;
7.              while (turn != id) {
8.                  while (blocked[1-id]) {
9.                      /* do nothing – busy wait loop */
10.                 }
11.                 turn = id;
12.             }
13.             /* line 14: Code for critical section */
14.             ...
15.             /* line 16: Exit from critical section code */
16.             blocked[id] = false;
17.             /* remainder code */
18.             ...
19.         }
20.     }
21.
22.     void main() {
23.         blocked[0] = false;
24.         blocked[1] = false;
25.         turn = 0;
26.         /* Create 2 parallel threads,
27.          * one executes P(0),
28.          * the other executes P(1)
29.          */
30.         ...
31.     }
```

Context switch occurs at this point

Example showing execution of threads:

```
      Thread T0              Thread T1
      ------------------------------------
|       Line N1
|       Line N2
|                  switch
|                            Line M1
|                            Line M2
|                  switch
|       Line N3
v       Line N4
Time
```

You do not have to follow this model exactly, but be clear about what each thread executes, and in what order.

Give an execution sequence that shows this solution does not satisfy mutual exclusion. Indicate the lines executed by each thread, and where context switches occur.

Thread 0		Thread 1
		Line 10: T1 runs until just before Line 11
	switch	
Line 4,5,6,7,14 enter critical section		
	switch	
		Line 11,7,14 enter critical section

Question 3. Synchronization [10 MARKS]

You are implementing an OS with a partner. You ask your partner to write some synchronization functions:

lock(L)	Locks L.
unlock(L)	Unlocks L.
sleep()	Makes the current thread sleep until woken with wake().
wake(T)	Wakes thread T.

Your partner writes the functions above, and then, being really smart, decides to write several combination functions:

lock_sleep(L)	Atomically lock(L) and then sleep().
unlock_sleep(L)	Atomically unlock(L) and then sleep().
lock_wake(L, T)	Atomically lock(L) and then wake(T).
unlock_wake(L, T)	Atomically unlock(L) and then wake(T).

You may assume that all of these functions, but no others, execute atomically.

Part (a) [2 MARKS] You wish to implement the P() and V() semaphore operations using the atomic operations above. Clearly explain why the original synchronization functions (the non-combination functions) are not sufficient for implementing the semaphore operations.

There is no atomic primitive for doing an unlock and sleep.

The original synchronization functions will either require sleeping with locks held, causing a potential deadlock, or require releasing the lock before sleeping, causing a potential lost wakeup.

Part (b) [8 MARKS] Implement the semaphore operations using the atomic operations above. Clearly show the shared variables that your code uses. You can assume that a queue implementation is available to you. Clearly indicate the queue functions that your code uses and the type of the arguments provided to the queue functions.

```
// shared variables

Lock L;
int V = 0; // Semaphore value
Queue Q; // queue of waiting threads



// queue functions used and their arguments

void add(Queue Q, Thread t); // add to queue, the thread t
Thread *remove(Queue Q); // remove a thread from the queue
boolean is_empty(Queue Q); // is the queue empty?


P() {

    lock(L);
    if (V == 0) {
        add(Q, cur_thread());
        unlock_sleep(L);
        lock(L);
    }
    V = V - 1;
    unlock(L);


}

V() {

    lock(L);
    if (!is_empty(Q)) {
        wake(remove(Q));
    }
    V = V + 1;
    unlock(L);


}
```

Question 3. Synchronization [10 MARKS]

A user has written the snippet of code shown below. Assume that a main thread (code not shown) invokes `Init_thread` once and `Worker_thread` multiple times. All these threads are invoked concurrently.

Use blocking locks and condition variables to ensure that there are no races in this code and to ensure two conditions: 1) the initialization code ($x = 0$) in `Init_thread` runs before x is incremented ($x = x + 1$) by any `Worker_thread`, 2) after 10 worker threads have each incremented x, `Init_thread` calls `exit` so that the program exits.

Make sure to clearly declare and initialize any synchronization variables. You will not need any additional non-synchronization variables. Since you are using blocking synchronization primitives, do not use `sleep`, `wakeup`, tight loops, or interrupt disabling.

```
int x = -1;
lock l = unlocked;
cv init;
cv end;


Init_thread() {
        lock(l);



        x = 0;
        signal(l, init);
        wait(l, end);
        unlock(l);

        exit(0);


}

Worker_thread() {
        lock(l);
        while (x == -1) {
          wait(l, init);
        }
        x = x + 1;
        if (x == 10)
          signal(l, end);
        unlock(l);

}
```

A similar check can also be placed in Init_thread().

Question 4. Multiprocessor Synchronization [20 MARKS]

We have seen how the semaphore primitives P and V are implemented for uniprocessors. In this problem, we will implement these primitives for multiprocessors. The relevant code for the uniprocessor version of these primitives is shown below. The `schedule()` function removes the current thread from the run queue and chooses another thread to run.

```
P(sem) {
    spl = splhigh();
    while (sem->count==0) {
        thread_sleep(sem);
    }
    sem->count--;
    splx(spl);
}
```

```
V(sem) {
    spl = splhigh();
    sem->count++;
    thread_wakeup(sem);
    splx(spl);
}
```

```
thread_sleep(void *cond) {
    add_to_wait_queue(cond);
    schedule();
}
```

```
thread_wakeup(void *cond) {
    thread = remove_from_wait_queue(cond);
    add_to_run_queue(thread);
}
```

Part (a) [16 MARKS] In the four questions shown below, circle all (one or more) correct answers. Be careful, two marks will be deducted for each incorrect answer. In the answers below, “corrupt” means an incorrect update.

1) Would the semaphore code shown above work for multiprocessors?

[✓ 1.] No, it could corrupt the `sem->count` variable.

[✓ 2.] No, it could corrupt the wait queue.

[✓ 3.] No, it could corrupt the run queue.

[✓ 4.] No, it could cause a thread to wait forever.

5. Yes, it would work.

2) Your partner argues that the code would clearly not work for multiprocessors and suggests a minimal change: add spin locks in the `thread_sleep` and `thread_wakeup` implementation:

```
thread_sleep(void *cond) {
    spin_lock(schedlock);
    add_to_wait_queue(cond);
    schedule();
    spin_unlock(schedlock);
}
```

```
thread_wakeup(void *cond) {
    spin_lock(schedlock);
    thread = remove_from_wait_queue(cond);
    add_to_run_queue(thread);
    spin_unlock(schedlock);
}
```

Would the semaphore code shown earlier now work for multiprocessors?

[✓ 1.] No, it could corrupt the `sem->count` variable.

2. No, it could corrupt the wait queue.

3. No, it could corrupt the run queue.

[✓ 4.] No, it could cause a thread to wait forever.

5. Yes, it would work.

wait queue and run queue have no races now but P and V still have a race.

3) Your friend from Barkly University argues that the code would clearly still not work for multiprocessors and suggests another minimal change: replace interrupt disabling with spin locks in the `P` and `V` implementation:

```
P(sem) {
    spin_lock(sem->spinlock);
    while (sem->count==0) {
        thread_sleep(sem);
    }
    sem->count--;
    spin_unlock(sem->spinlock);
}
```

```
V(sem) {
    spin_lock(sem->spinlock);
    sem->count++;
    thread_wakeup(sem);
    spin_unlock(sem->spinlock);
}
```

Would this semaphore code work for multiprocessors?

1. No, it could corrupt the `sem->count` variable.

2. No, it could corrupt the wait queue.

3. No, it could corrupt the run queue.

[✓ 4.] No, it could cause a thread to wait forever.

5. Yes, it would work.

spin_lock is not released in P when sleeping. deadlock is possible.

4) Your partner looks at the code, mutters something about Barkly, and suggests adding `spin_unlock` and `spin_lock` around `thread_sleep` in the `P` implementation:

```
P(sem) {
    spin_lock(sem->spinlock);
    while (sem->count==0) {
        spin_unlock(sem->spinlock);
        thread_sleep(sem);
        spin_lock(sem->spinlock);
    }
    sem->count--;
    spin_unlock(sem->spinlock);
}
```

unlock done before sleep. wake up be lost, causing a thread to wait forever.

sem->count is protected by sem->spinlock, so it will not be corrupted.

You are convinced this code will not work for multiprocessors?

1. No, it could corrupt the `sem->count` variable.

2. No, it could corrupt the wait queue.

3. No, it could corrupt the run queue.

[✓ 4.] No, it could cause a thread to wait forever.

Part (b) [4 MARKS] You know that the previous code is almost correct, but it does cause problems occasionally. How would you go about fixing the problem? Use the bullet points shown below to state your solution. Be as precise as possible. Hint: the implementation of the semaphore primitives should be tightly coupled with the sleep/wakeup primitives.

1. Add a new spinlock parameter to thread_sleep.

2. Move spin_unlock(sem->spinlock) from the P() code to right AFTER spin_lock(schedlock). In this case, wake up will not be lost because unlocking the semaphore spin lock is done after the scheduler spin lock is acquired (and wakeup will block on acquiring the scheduler lock).

3.

4.

3 Monitors and Semaphores [10 points]

Your task is to implement semaphores using monitors. Assume that your programming environment supports monitors (e.g., it provides condition variables, etc.). Write a monitor called Semaphore that implements the functionality of the Down and Up semaphore methods. You are not allowed to disable interrupts in your code. An outline of the code is shown below:

Part A [8 points]

If your implementation makes any assumptions, state them.

```
Monitor Semaphore {




};

Semaphore::Down() {




}

Semaphore::Up() {




}
```

Part B [2 points]

Show an example of how the Down() method should be invoked.

2 Synchronization [10 points]

A user has written the snippet of code shown below. Assume that `Init_thread` is invoked once while `Worker_thread` can be invoked one or more times concurrently. Write code to ensure that there are no races in this code. Your code should also ensure that the initialization code in the `Init_thread` (`x = 10`) runs before `x` is incremented in any `Worker_thread`. If you use any variables in your code, make sure that they are declared and initialized correctly.

Answers shown in bold, using condition variables and semaphores.

```
int x;
int initialized = 0;
lock l = 0; // unlocked;
condition c;

Init_thread() {
    lock(l);
    x = 10;
    initialized = 1;
    signal(c);
    unlock(l);
}

Worker_thread() {
    lock(l);
    while (initialized == 0)
        wait(c, l);
    x = x + 1;
    unlock(l);
}
```

```
int x;
semaphore sem = 0;
Init_thread() {
    x = 10;
    up(sem);
}

Worker_thread() {
    down(sem);
    x = x + 1;
    up(sem);
}
```

4 Synchronization [10 points]

A group of workers need to carry boxes across a bridge. The bridge is weak, so only two workers can go across the bridge at a time. The boxes are heavy, so the two workers must carry one box at a time. There should be no unnecessary waiting, i.e., if two workers and one box is available, the bridge should be used.

The code below shows a semaphore-based implementation for this problem. A worker calls the procedure WorkerReady when she is ready to use the bridge. When a box is ready to be moved across the bridge, the procedure BoxReady is called. Together, these procedures provide one box to two workers and then call EnterBridge, after which the two workers carry the box across the bridge. Assume that the EnterBridge procedure correctly blocks the caller until the bridge has been crossed, and enough boxes are available for each worker. Assume also that the semaphore waiting queues are FCFS.

```
semaphore worker_ready = 0;
semaphore box_ready = 0;

void                                 void
WorkerReady()                        BoxReady()
{                                    {
    worker_ready.up();                   worker_ready.down();
    box_ready.down();                    worker_ready.down();
}                                        EnterBridge();
                                         box_ready.up();
                                         box_ready.up();
                                     }
```

State whether the implementation works (it is correct) or does not work (it is either always wrong or there exists an execution scenario where it is wrong). Then justify your answer by either proving that the implementation is correct, OR showing an execution scenario where the implementation is not correct and suggest a way to fix it.

The implementation does not work correctly.

Consider the case when two workers are ready to cross the bridge. They both wait at box_ready.down(). Now suppose two boxes become ready, and BoxReady() is invoked for each box. Each box could issue worker_ready.down() and then both boxes would wait on the second worker_ready.down(). In this case, even though two workers and a box are available, EnterBridge() is not called, so there is unnecessary waiting.

The fix consists of introducing a new lock that makes the two calls to worker_ready.down() a critical section. In that case, when a box arrives, it will wait for two workers and then be able to call EnterBridge.

2. Number Dispenser [12 marks]

John has been asked to design a number dispenser for the driver's license office. This office has N counters. A person arriving at the license office, clicks a button on the dispenser and then **waits** until the machine provides the person with a counter number (CN). The person then goes to the CN counter to renew his or her license. When the person leaves the counter, the agent lets the dispenser know that the counter is available.

John designs his number dispenser with two procedures, `find_counter()` and `leave_counter()` as shown below. These procedures are invoked in a separate thread for each person. `available[i]` is set to TRUE if counter `i` is available. `num_people` is a global variable that counts the total number of people being served by the agents at the counter.

```
// invoked when person clicks number dispenser button
int
find_counter()
{
    int i;
    while (1)  {
        for (i = 0; i < N; i++) {
            if (available[i] == TRUE) {
                available[i] = FALSE;
                num_people = num_people + 1;
                return i; // returns CN
            }
        }
    }
}

// invoked by agent
void
leave_counter(int TCN)
{
    available[i]= TRUE;
    num_people = num_people – 1;
}
```

John's code will be correct : 1) if an agent serves at most one person at a time, 2) if `num_people` correctly counts the number of people being served by different agents, and 3) if an agent is free, and no person invokes `find_counter()` or `leave_counter()`, then a person invoking `find_counter()` should be able to get a counter number.

John runs his code and finds that as long as one person arrives at a time, his code works fine. However, when people come concurrently, the dispenser gets confused.

a) [4 marks] Circle the true statements below. ½ mark will be deducted for wrong answers.

[✓ A.] There is a race condition in accesses to `available[]` when two `find_counter()` threads run.

B. . There is a race condition in accesses to `available[]` when one `find_counter()` and one `leave_counter()` thread runs.

[✓ C.] There is a race condition in accesses to `num_people` when more than one thread updates `num_people.`

D. There is no race condition as long as the minimum time between people requesting a counter number is larger than the maximum time needed to run `find_counter()`.

Version 1:

John is encouraged by his friend to take ECE353. He learns about spin locks and changes his code as follows (changes shown in bold):

```
// invoked when person clicks number dispenser button
int
find_counter()
{
    int i;
    while (1)  {
        spin_lock(counter_lock);
        for (i = 0; i < N; i++) {
            if (available[i] == TRUE) {
                available[i] = FALSE;
                num_people = num_people + 1;
                return i; // returns CN
            }
        }
    }
}

// invoked by agent
void
leave_counter(int TCN)
{
    available[i]= TRUE;
    num_people = num_people – 1;
    spin_unlock(counter_lock);
}
```

b) [2 marks] Does this code work correctly? Why or why not?

No. Doesn't satisfy correctness criterion 3.

Version 2:

Hoping to improve his code, John writes another version:

```
// invoked when person clicks number dispenser button
int
find_counter()
{
    int i;
    while (1)  {
        spin_lock(counter_lock);
        for (i = 0; i < N; i++) {
            if (available[i] == TRUE) {
                available[i] = FALSE;
                num_people = num_people + 1;
                spin_unlock(counter_lock);
                return i; // returns CN
            }
        }
        spin_unlock(counter_lock);
    }
}

// invoked by agent
void
leave_counter(int TCN)
{
    spin_lock(counter_lock);
    available[i]= TRUE;
    num_people = num_people – 1;
    spin_unlock(counter_lock);
}
```

c) [2 marks] Does this code work correctly? Why or why not?

Yes. Code ensures mutual exclusion of available and num_people and satisfies criteria 3 above.

Version 3:

Hoping to reduce contention, John rewrites the `find_counter()` as follows:

```
// invoked when person clicks number dispenser button
int
find_counter()
{
    int i;
    while (1)  {
        for (i = 0; i < N; i++) {
            spin_lock(counter_lock);
            if (available[i] == TRUE) {
                available[i] = FALSE;
                num_people = num_people + 1;
                spin_unlock(counter_lock);
                return i; // returns CN
            } else {
                spin_unlock(counter_lock);
            }
        }
    }
}
```

d) [2 marks] Does this code work correctly? Why or why not?

Yes. Code ensures mutual exclusion of available and num_people and satisfies criteria 3 above.

e) [2 marks] Suppose that there are always many people waiting to get a number at the license office. For which versions of the dispenser, can a person wait forever? Circle one or more below. ½ mark will be deducted for wrong answers.

[✓ A.] Version 1

[✓ B.] Version 2

[✓ C.] Version 3

3. Synchronization [6 marks]

Canada and US are separate threads executing their respective procedures shown below. The code below is intended to cause them to forever take turns exchanging insults through the shared variable X in strict alternation. The Sleep() routine blocks the calling thread, and the Wakeup() routine unblocks a specific thread if that thread is blocked.

```
void
Canada ()
{
  while (1) {
    Sleep();
    X = ShoutInsult(X);
    Wakeup(USThread);
  }
}
```

```
void
US()
{
  while (1) {
    X = ShoutInsult(X);
    Wakeup(ThreadCanada);
    Sleep();
  }
}
```

a) [2 marks] The code shown above exhibits a well-known synchronization flaw. Outline a scenario in which this code would fail, and the outcome of that scenario. What is this flaw called?

Canada is about to go to Sleep(). US() performs Wakeup() and then goes to Sleep(). Canada goes to Sleep(). Both sleep forever. This is the lost wakeup problem.

b) [4 marks] Show how to fix the problem using semaphores, replacing the Sleep() and Wakeup() calls with semaphore Wait (P/down) and Signal (V/up) operations. Note: Disabling interrupts is not an option.

```
Semaphore US, Canada = 0;

Canada()
{
   while(1) {
      Wait(US);
      X = ShoutInsult(X);
      Signal(Canada);
   }
}

US()
{
   while (1) {
      X = ShoutInsult(X);
      Signal(US);
      Wait(Canada);
   }
}
```

3. My App Doesn’t Shut Down Cleanly [10 marks]

The following simplified code snippet is taken from the Apache web server. The design is as follows:

1. Apache has one worker thread that executes the `run()` function. It executes mainly within the `while(!stopped)` loop.
2. When Apache needs to shut down, a second thread will execute `doStop()`.
3. `doStop()` forces the worker thread to exit the `sock.accept` function; it then sets `stopped` to be TRUE and waits for the worker thread's acknowledgement.
4. When the worker thread reads a TRUE value for `stopped`, it breaks out of the loop and notifies the `doStop` thread using a condition variable.

```
/* These 4 global variables are initialized correctly */

 1 pthread_mutex_t mutex;
 2 pthread_cond_t cond;
 3 ServerSocket sock;
 4 int stopped = FALSE;

 5 void doStop() {
 6   sock.cancel();
 7   stopped=TRUE;

 8   pthread_mutex_lock(&mutex);
 9   pthread_cond_wait(&cond, &mutex);
10   pthread_mutex_unlock(&mutex);
        ...
   }

11 void run() {
     ...
12   while (!stopped) {
13     sock.accept();
       /* sock.accept() returns when either
        * (1) it receives a request OR,
        * (2) another thread executes sock.cancel
        * during the sock.accept() */
        ...
      }
14    pthread_mutex_lock(&mutex);
15    pthread_cond_signal(&cond);
16    pthread_mutex_unlock(&mutex);
   }
```

Shortly after the code was released, users began to complain that their Apache servers would sometimes hang during shutdown (that is, the shutdown would not complete).

A) [4 marks] There are two different problems with the code that can cause a ``hang" to occur in different places. Give two different execution scenarios that illustrate these two problems. You can make the following assumptions: (i) There are exactly 2 threads - one executing `doStop` and one executing `run`. (ii) No new client requests are received after the user starts the shutdown.

1. doStop() is preempted after calling sock.cancel() and before stopped = TRUE is run. run() comes out of sock.accept() as a result of sock.cancel(), checks and finds that stopped is not TRUE and goes back to sock.accept() and waits there forever.

2. run() issues pthread_cond_signal() before doStop() issues pthread_cond_wait().

B) [6 marks] Correct the code. The following shows the original code. You can make any changes to it,

```
/* Add any new global variables */

   int signaled = 0;

 void doStop() {

   ~~sock.cancel();~~

   stopped=TRUE;
     sock.cancel();

   pthread_mutex_lock(&mutex);
```

```
      if (signaled == 0) {
 9    pthread_cond_wait(&cond, &mutex);
      }
10    pthread_mutex_unlock(&mutex);

   }

11 void run() {
      ...

12    while (!stopped) {

13      sock.accept();
        ...

      }

14    pthread_mutex_lock(&mutex);
      signaled = 1;
15    pthread_cond_signal(&cond);

16    pthread_mutex_unlock(&mutex);

   }
```

signaled can be either before or after pthread_cond_signal

4. Barriers [8 marks]

A barrier is a type of thread synchronization that ensures all threads in a group arrive at the same instruction before any of them proceed. A thread waits at a barrier until all threads have performed a wait at the barrier. They then can all proceed. You wonder whether the previous Apache shutdown problem could be easily solved using barriers, and decide to implement a barrier using condition variables and locks.

The prototypes of the condition variable and lock functions you implemented in OS161 are provided below.

```
struct lock *lock_create(void);
void         lock_acquire(struct lock *);
void         lock_release(struct lock *);
void         lock_destroy(struct lock *);

struct cv *cv_create(void);
void       cv_wait(struct cv *cv, struct lock *lock);
void       cv_signal(struct cv *cv, struct lock *lock);
void       cv_broadcast(struct cv *cv, struct lock *lock);
void       cv_destroy(struct cv *);
```

4. Barriers [8 marks]

A barrier is a type of thread synchronization that ensures all threads in a group arrive at the same instruction before any of them proceed. A thread waits at a barrier until all threads have performed a wait at the barrier. They then can all proceed. You wonder whether the previous Apache shutdown problem could be easily solved using barriers, and decide to implement a barrier using condition variables and locks.

The prototypes of the condition variable and lock functions you implemented in OS161 are provided below.

```
struct lock *lock_create(void);
void         lock_acquire(struct lock *);
void         lock_release(struct lock *);
void         lock_destroy(struct lock *);

struct cv *cv_create(void);
void       cv_wait(struct cv *cv, struct lock *lock);
void       cv_signal(struct cv *cv, struct lock *lock);
void       cv_broadcast(struct cv *cv, struct lock *lock);
void       cv_destroy(struct cv *);
```

a) We have shown some sample barrier code below. Add any code in the spaces provided below, as required, to implement barriers.

```
struct barrier {
 struct lock *mutex;
 struct cv *cond;
  int n_total_th;
  int n_arrived_th;

};

/* num_threads: number of threads that will wait at barrier */
struct barrier *
barrier_create(int num_threads) {
  struct barrier *b;
  b = (struct barrier *)malloc(sizeof(struct barrier));
  b->mutex = lock_create();
  b->cond = cv_create();
  b->n_total_th = num_threads;
  b->n_arrived_th = 0;

  return b;
}

void
barrier_wait(struct barrier * b) {

lock_acquire(b->mutex);
  b->n_arrived_th++;
  if (b->n_arrived_th < b->n_total_th) {
     cv_wait(b->cond, b->mutex);
  }
  else
  {
     cv_broadcast(b->cond);
     // Prepare for the next use of barrier_wait.
     b->n_arrived_th = 0;
  }

  lock_release(b->mutex);

}

void
barrier_destroy(struct barrier * b) {


  lock_destroy(b->mutex);
  cv_destroy(b->cond);
  free(b);
}
```

6. My First 32 Core Multi-Processor [8 marks]

You just bought your first 32 core SMP machine. You have learnt that the test-and-set instruction is quite expensive (as compared to a normal load from memory, for example). Thus, you have decided to use this code to implement a lock in OS161:

```
typedef struct __lock_t {
  int flag; // init to 0
} lock_t;

int TestAndSet(int *lock, int new) {
  int old = *lock;
  *lock = new;
  return old;
}

void lock(lock_t *lock) {
  do {
    while (!lock->flag); // unprotected lock check, spin
  } while (TestAndSet(&lock->flag, 1)); // actual atomic locking
}

void unlock(lock_t *lock) {
  lock->flag = 0;
}
```

A) [2 marks] Does this lock work correctly? Why or why not? If not, how would you fix it?

No.

while (lock->flag); // condition is inverted.

B) [2 marks] Assuming the code above is correct or you have fixed it, when does it perform better than a simple spin lock built with test-and-set? When does it perform worse?

Better when there is lock contention.

Worse when there is none, because of the extra unprotected lock check.

C) [2 marks] You feel unsafe about the unlock code. So you change it as shown below?

```
void unlock(lock_t *lock) {
  TestAndSet(&lock->flag, 0);
}
```

Does this unlock work correctly? (why or why not?)

Yes. It does the same thing as setting flag to 0.

D) [2 marks] Does this unlock perform better, worse, or the same as the earlier implementation?

It is slower because it uses an atomic instruction.

Question 1. Concurrency [4 MARKS]

Suppose two threads execute the following C code concurrently, accessing shared variables `a`, `b`, and `c`:

```
// initialization code
int a = 4;
int b = 0;
int c = 0;

// Thread 1                    // Thread 2

if (a < 0) {                   b = 10;
  c = b - a;                   a = -3;
} else {
  c = b + a;
}
```

What are the possible values for `c` after both threads complete? You can assume that reads and writes of the variables are atomic, and that the order of statements within each thread is preserved in the code generated by the C compiler. Marks will be deducted for wrong or too many answers.

13

4

14

7

Question 2. Synchronization [8 MARKS]

To make laughing gas, two Nitrogen (N) atoms and one Oxygen (O) atom have to be fused. Let us synthesize this reaction by treating the atoms as threads and synchronizing them.

Each N atom invokes a procedure nReady() when it is ready to react, and each O atom invokes a procedure oReady() when it is ready. The procedures delay until there are at least two N atoms and one O atom present, and then the oReady() procedure calls a Laugh() procedure. After the Laugh() call, the two instances of nReady() and one instance of oReady() return.

To be fair, the N and the O atoms must fuse in the order they arrive. For example, N1 and N2 should fuse with O1, N3 and N4 should fuse with O2, and so on.

N

The code below shows a proposed solution for creating laughing gas. The count variable tracks the order in which the O atoms arrive. You may assume that the semaphore implementation enforces FIFO order for wakeups, i.e., the thread waiting longest in P() is always the next thread woken up by a call to V().

```
Semaphore nwait = 0;
Semaphore owait = 0;
int count = 0;

nReady()                              oReady()
{                                     {
  count++;                              P(owait);
  if (count % 2 == 1) {                 Laugh();
    P(nwait);                           V(nwait);
  } else {                              V(nwait);
    V(owait);                           return;
    P(nwait);                         }
  }
  return;
}
```

Part (a) [2 MARKS] Unfortunately, this code has a race. Describe a case in which this code will not work.

The race is in the update of count. Multiple N atoms share count, and so count needs to be updated in a critical section.

Part (b) [6 MARKS] Below, you will try to fix the race. We have added the `lock()` code below. You need to add `unlock()` in the code below. Write within each box shown below, **one** of the following: 1) **unlock**, if the code needs an unlock there, 2) **deadlock**, if adding an unlock there may cause a deadlock, and 3) **incorrect**, if adding an unlock there may cause any other kind of incorrect synchronization.

```
Semaphore hwait = 0;
Semaphore owait = 0;
int count = 0;

Lock L;     // new code

nReady()
{
  lock(L); // new code
  count++;
                                  ⟸  deadlock: 3 N threads update count to 3, and then
                                      all will get stuck in P(nwait)
  if (count % 2 == 1) {
                                  ⟸  unlock(L);
    P(nwait);
                                  ⟸  deadlock: after 1st N thread waits at P(), no one
                                      else gets to run.
  } else {
                                  ⟸  incorrect: N thread 2 and 3 could fuse with O
                                      thread 1. However, we also accepted an unlock
                                      here as acceptable answer.
    V(owait);
    P(nwait);
                                  ⟸  unlock(L);
  }
                                  ⟸  deadlock: after 1st N thread waits at P(), no one
                                      else gets to run.
  return;
}
```

Question 4. Red-Blue Race [10 MARKS]

Car owners are trying to decide whether red cars or blue cars are faster. They device an experiment in which all the red cars line up behind each other in a red line, and similarly all the blue cars line up behind each other in a blue line. Your job is to ensure that a red car and a blue car at the head of the two lines start the race at about the same time, i.e., either car should not start the race much before the other. You also need to ensure that the experiment makes progress as long as cars are present in both lines.

Part (a) [2 MARKS] You ask your friend John for help, and the pseudocode shown below is the best that he can come up with. A red car, upon arriving in the red line, invokes the function `red_lineup()`. When the function returns, the car starts racing. The blue solution is symmetric with the red solution (swap all occurrences of red and blue), and is not shown. Assume that `signal()` wakes up threads in the order they issued the `wait()`.

```
red_lineup()
{
        lock(lock);
        red_waiting++;
        while (blue_waiting == 0) {
           wait(red, lock);
        }
        signal(blue, lock);
        red_waiting--;
        unlock(lock);
}
```

You think about this solution and realize that it doesn't work correctly. Explain why.

red arrives and waits
blue arrives and signals, reduces blue_waiting to 0
red still waits because of "while"

This question continues
on the following page.

Part (b) [3 MARKS] Now you ask your friend Mary for help, and the pseudocode shown below is the best that she can come up with.

```
red_lineup()
{
        lock(lock);
        red_waiting++;
        if (blue_waiting == 0) {
           wait(red, lock);
        } else {
           signal(blue, lock);
        }
        red_waiting--;
        unlock(lock);
}
```

You think about this solution and realize that it still doesn't work correctly. Explain why.

red arrives and waits
blue arrives, signals, exits
second blue arrives, should wait, but continues because red_waiting is still 1

Part (c) [5 MARKS] You think hard about why John and Mary's solutions didn't work and realize that a small change will make the solution work. Show that solution below.

the problem is that in part b), when the first blue signals, it should have reduced red_waiting as well.

```
red()
{
    lock(lock);
    red_waiting++;
    if (blue_waiting == 0) {
        wait(red, lock);
    } else {
        red_waiting--;
        blue_waiting--;
        signal(blue, lock);
    }
    unlock(lock);
}
```

Question 5. Deadlocks [10 MARKS]

Part (a) [5 MARKS] Alice, Bob, and Carol go to a Chinese restaurant at a busy time of the day. The waiter apologetically explains that the restaurant can provide only two pairs of chopsticks (for a total of four chopsticks) to be shared among the three people. Alice proposes that all four chopsticks be placed in an empty glass at the center of the table and that each diner should obey the following protocol:

```
while (!had_enough_to_eat()) {
    acquire_one_chopstick(); /* May block. */
    acquire_one_chopstick(); /* May block. */
    eat();
    release_one_chopstick(); /* Does not block. */
    release_one_chopstick(); /* Does not block. */
}
```

Can this dining plan lead to deadlock? Explain your answer.

Deadlock **cannot** occur because there are enough chopsticks to guarantee that at least one of the 3 diners will be able to get the 2 chopsticks that he/she needs. Once that diner finishes, he/she will release their chopsticks for someone else to use, so eventually everyone finishes.

Part (b) [5 MARKS] Suppose now that instead of three diners there will be an arbitrary number, D. Furthermore, each diner may require a different number of chopsticks to eat. For example, it is possible that one of the diners is an octopus, who for some reason refuses to begin eating before acquiring eight chopsticks. The second parameter of this scenario is C, the number of chopsticks that would simultaneously satisfy the needs of all diners at the table. For example, Alice, Bob, Carol, and one octopus would result in C = 14. Each diner's eating protocol is shown:

```
int s;
int num_sticks = my_chopstick_requirement();
while (!had_enough_to_eat()) {
    for (s = 0; s < num_sticks; ++s) {
        acquire_one_chopstick(); /* May block. */
    }
    eat();
    for (s = 0; s < num_sticks; ++s) {
        release_one_chopstick(); /* Does not block. */
    }
}
```

What is the smallest number of chopsticks (in terms of D and C) needed to ensure that deadlock cannot occur? Explain your answer.

C – D + 1. This guarantees that every diner can get all but one of the chopsticks it needs (C – D), with one additional chopstick to guarantee that at least one diner gets all of the chopsticks it needs.

Question 2. Thread Switching [5 MARKS]

You know that implementing thread switching is a little tricky. To simplify matters, you start by implementing thread switching between just two threads, A and B, as shown below. You have added A and B to the run queue, and initialized their thread context so that they will start executing the `thread_A()` and `thread_B()` functions when they are run for the first time. Your scheduler runs Thread A first.

```
int i = 0;
ucontext_t uA, uB;

thread_A() {                          thread_B() {
    int d = 0;                            int d = 1;
    while (i < 3) {                       while (i < 3) {
        i++;                                  i++;
        printf("A:%d␣", i);                   printf("B:%d␣", i);
        d = 0;                                d = 1;
        getcontext(&uA);                      getcontext(&uB);
        if (d == 0) {                         if (d == 1) {
            d = 1;                                d = 0;
            setcontext(&uB);                      setcontext(&uA);
        }                                     }
    }                                     }
}                                     }
```

Part (a) [3 MARKS] Circle the output you expect to see? Hint: Think carefully about what is saved by the `getcontext()` function.

A) A:1 A:2 A:3

B) A:1 B:2 B:3

C) A:1 B:2 A:3 B:4 A:5 B:6

D) A:1 B:2

E) A:1 B:1

[✓ F) A:1 B:2 A:3]

We gave 3 marks for a correct answer.

For an incorrect answer, we gave 1 partial mark for Part(a) if Part(b) gave a reasonable explanation for the choice here.

Part (b) [2 MARKS] Briefly explain why you have chosen the answer above.

The main idea that we expected students to understand is that the getcontext call saves the stack pointer and not the values of the stack variables. Execution happens as follows:
print A:1, a:d=0, a:getcontext, a:d=1, a:setcontext, print B:2, b:d=1,b:getcontext,b:d=0,b:setcontext
now we start running a at the point its getcontext was called. The value of a:d is 1 (note that it was set after the getcontext call), so we go to the next iteration of a's loop.
print A:3, a:d=0, a:getcontext, a:d=1,a:setcontext
now we start running b at the point its getcontext was called. The value of b:d is 0, so we go to the next iteration of b's loop. However, i is 3 at this point, so b stops. Thread a nevers runs again.

Question 3. Synchronization [5 MARKS]

Consider the following program that uses three threads (A, B and C). It synchronizes these threads using locks and condition variables. The program begins by invoking the `thread_A()` function.

```
struct lock *l;
struct cv *cv;

void                                      void
thread_A()                                thread_B(void *arg)
{                                         {
        l = lock_create();                        lock_acquire(l);
        cv = cv_create();                         cv_signal(cv, l);
        thread_create(thread_B, 0);               thread_yield(THREAD_ANY);
        thread_create(thread_C, 0);               cv_wait(cv, l);
        thread_yield(THREAD_ANY);                 lock_release(l);
        thread_yield(THREAD_ANY);         }
        printf("STOP␣HERE\n");
}                                         void
                                          thread_C(void *arg)
                                          {
                                                  lock_acquire(l);
                                                  cv_wait(cv, l);
                                                  thread_yield(THREAD_ANY);
                                                  cv_signal(cv, l);
                                                  lock_release(l);
                                          }
```

Part (a) [3 MARKS] Trace the execution of this program until it prints out the message "STOP HERE" by writing down the sequence of context switches that have occurred up to this point. Assume that the scheduler runs threads in FIFO order with no time-slicing (non-preemptive scheduling), all threads have the same priority, and threads are placed in wait queues in FIFO order. The output shown below should be in the form $B \rightarrow A \rightarrow C$, signifying that thread B context switches to thread A, which then context switches to thread C. Hint: Think carefully about how the condition variable primitives work.

A ----------> B ----------> C ----------> A ----------> B ----------> A

yield yield acquire yield wait STOP HERE

Note that when B issues a wait(), it releases the lock, which wakes up C, but C is added to the back of the ready queue after A, so C never gets to run again.

Part (b) [2 MARKS] When the program prints out the message "STOP HERE", list the currently running thread, and the (zero or more) threads in the ready and the wait queues shown below.

current thread: A

ready queue: C

lock wait queue:

cv wait queue: B

Question 4. **Waiting for Exit** [5 MARKS]

You have implemented an early version of the Unix operating system that provides the signal facility (e.g., the `kill` system call for sending signals). Your operating system also provides the `sleep` system call, which blocks a process until a signal is sent to the process.

Your operating system, however, does not provide the `wait` system call to allow a parent to wait until its child process exits. You consider implementing the functionality of the `wait` system call at the user level with the code shown below. The initial call to `signal` registers an empty signal handler. The `sleep` call blocks until a signal is delivered, after which it returns.

```
void
null(int unused)
{
}

int
main()
{
        int pid;

        signal(SIGUSR1, null);
        pid = fork();
        if (pid) {
                /* block until child sends signal */
                sleep();
                printf("child␣is␣dead\n");
                exit(0);
        } else {
                kill(getppid(), SIGUSR1);
                exit(0);
        }
}
```

Describe two reasons why the code above does not implement the Unix `wait` functionality correctly.

1) 1. Kill can occur before the sleep is issued, so the parent may sleep forever. This is similar to the lost wakeup problem.

2. At the point, "child is dead" is printed, the child may not have exited yet.

3. Parent doesn't get child's exit value.

4. Some other signal sent to the parent will wakeup sleep, even though child hasn't issued kill.

2)

Question 2. Atomic or Not [6 MARKS]

The fetch-and-add hardware instruction *atomically* increments a value while returning the old value at a particular address. The C pseudocode for the fetch-and-add instruction looks like this:

```
int fetch_and_add(int *ptr)
{
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

The fetch-and-add instruction can be used to build a *ticket lock*, as shown below. When a thread wishes to acquire a lock, it first does an atomic fetch-and-add on the ticket value. The returned value is considered this thread's "turn" (myturn). The shared value `l->turn` is then used to determine whether this thread can acquire the lock. When `myturn == l->turn` for a given thread, it is that thread's turn to enter the critical section. The lock is released by incrementing `l->turn` so that the next waiting thread (if there is one) can now enter the critical section.

```
struct lock {
    int ticket;
    int turn;
};

void lock_init(struct lock *l)
{
    l->ticket = 0;
    l->turn = 0;
}
```

```
void acquire(struct lock *l)
{
    int myturn = fetch_and_add(&l->ticket);
    while (myturn != l->turn)
        ; // spin
}

void release(struct lock *l)
{
    l->turn = l->turn + 1;
}
```

Part (a) [4 MARKS] Alice and Bob look at this code for a long time. Bob is convinced that the `release` code has a race (he suggests using `fetch_and_add` to increment `l->turn` to fix the race). Alice is convinced that the ticket lock code shown above is correct. Who is correct? Why?

Alice is correct. There is no race because l->turn is only updated by release(), and only the thread that calls acquire() calls release(). The acquire() function only reads the value of l->turn. It will either get the old or the new value of l->turn but this will not affect the correctness of the code.

Part (b) [2 MARKS] Assuming that Alice and Bob figure out how to implement the ticket lock correctly, would there be any benefit to using a ticket lock over a spin lock?

The ticket lock ensures fairness since each thread gets a ticket on arrival, unlike spinlocks. Also, ticket locks can be more efficient because they perform a regular read instruction instead of an atomic instruction while spinning in the acquire() code.

Question 3. My Way or the Highway [12 MARKS]

The Bloor Viaduct Bridge is undergoing repairs and only one lane is open for traffic. To prevent accidents, traffic lights have been installed at either end of the bridge to synchronize the traffic going in different directions. A car can only cross the bridge if there are no cars going in the opposite direction on the bridge. Sensors at either end of the bridge detect when cars arrive and depart from the bridge, and these sensors control the traffic lights. A skeleton implementation of two routines, `Arrive()` and `Depart()` is shown below. You may assume that each car is represented by a thread, and threads call `Arrive()` when they arrive at the bridge and `Depart()` when the leave the bridge. Threads pass in their direction of travel as input to the `Arrive()` routine.

```
int num_cars = 0;                         Lock *lock = new Lock();
enum dir = {open, east, west};            Condition *cv = new Condition();
dir cur_dir = open;

   void                                   void
   Arrive(dir my_dir)                     Depart()
   {                                      {

       lock(lock);                            lock(lock);
A1:    while (cur_dir != my_dir &&     D1:    num_cars--;
              cur_dir != open) {
              wait(cv, lock);
                                       D2:    if (num_cars == 0) {
       }

                                       D3:       cur_dir = open;
A2:    num_cars++;                               broadcast(cv, lock);

                                              }
A3:    cur_dir = my_dir;                      unlock(lock);
       unlock(lock);
                                          }
   }
```

Note that a broadcast(), not a signal() needs to be placed within the "if" statement, or else cars will get blocked.

Alternatively, a signal can be placed after the "if" statement, just before the "unlock"

Part (a) [4 MARKS] The code shown above does not do any synchronization. Show how two cars may travel in opposite directions at the same time. Use T1:E:A1 (or T1:W:A1) to indicate Thread T1 going East (or West) while executing statement A1.

T1:E:A1
T1:E:A2 (now num_cars has been incremented, but cur_dir has not been updated)
T2:W:A1 (another thread can enter because cur_dir is still "open")
T2:W:A2 ...

Part (b) [6 MARKS] Show how a lock and condition variable can be used to correctly synchronize the cars. Annotate the code above with calls to `lock()`, `unlock()`, `wait()`, `signal()`, and `broadcast()` operations (together with their arguments). Use only the operations that are needed or your answer will be penalized.

Part (c) [2 MARKS] Is there a problem with your solution? If so, give an example.

Even though the solution above synchronizes cars correctly, it can cause starvation. Cars going one way can starve cars going the other way (similar to reader-writer problem).