ECE 254 S15 Midterm Solutions

(1A) After the fork system call, an integer variable is returned by fork. This value is used to tell which is which. If it is the parent process, the value returned is the pid of the child; if it is 0, it means this is the child process.

(1B) A process is in the zombie state if it has finished execution but its parent has not yet collected its return value, so it cannot yet be cleaned up.

(1C)

```
void *search( void *void_arg ) {
    parameter_t *arg = (parameter_t *) void_arg;
    int *result = malloc( sizeof( int ) );
    *result = -1; // Default value

    for ( int i = arg->startIndex; i < arg->endIndex; ++i ) {
        if ( array[i] == arg->searchValue ) {
            *result = i;
            break;
        }
    }
    free( void_arg );
    pthread_exit(result);
}
```

(1D)

```
int main( int argc, char** argv ) {

    pthread_t threads[NUM_CPUS];
    void* returnValue;
    for ( int i = 0; i < NUM_CPUS; ++i ) {
        parameter_t* params = malloc( sizeof ( parameter_t ) );
        params->startIndex = i * (ARRAY_SIZE / NUM_CPUS);
        int end = (i + 1) * (ARRAY_SIZE / NUM_CPUS);
        if ( i == (NUM_CPUS - 1) ) {
            end = ARRAY_SIZE;
        }
        params->endIndex = end;
        params->searchValue = SEARCH_VALUE;

        pthread_create(&threads[i], NULL, search, params);
    }

    for ( int i = 0; i < NUM_CPUS; ++i ) {
        pthread_join( threads[i], &returnValue );
        int *rv = (int *) returnValue;
        if ( -1 != *rv ) {
            printf("Found at %d\n", *rv);
        }
        free( returnValue );
    }

    pthread_exit(0);
}
```

(2A) This one is intentionally somewhat tricky and requires holding to some precise definitions. In short: this solution is not vulnerable to deadlock, but it is vulnerable to starvation.

A scenario to cause the problem is what happens if the consumer thread tries to execute when the buffer is empty (count is 0). If it acquired the mutex it then spins forever in the while loop, checking if removed is false, but since it holds the mutex, the producer thread will be blocked on the wait statement and can never add it and never increment count.

An alternative scenario leading to a virtually identical problem is the producer runs exclusively, fills the buffer, and wants then to continue adding to the full buffer, but cannot. The buffer is full so added does not get changed to true. Once again, one process is in a never ending while loop while holding the mutex, never allowing the consumer to run to take items out of the buffer.

(Either one of these scenarios is acceptable in the justification of your answer. Or even a third valid scenario not listed here. Remember that for a solution to be unacceptable, we need only find one scenario that leads to problems.)

Deadlock requires that the processes are blocked permanently waiting for events that cannot come because they would need to be signalled by other blocked processes. If the producer (consumer) is in an infinite while loop and holds the mutex, it is not blocked and will appear, from the perspective of the operating system, to be executing and not blocked. Thus, this is not a deadlock.

It is, however, a case of starvation. If the producer (consumer) is stuck in the infinite while loop, the consumer (producer) will never be able to run to completion; it will wait infinitely to acquire the mutex. Thus, if the producer (consumer) is stuck like that, the consumer (producer) will starve.

(2B)

```
boolean testAndSet( int i ) {
  if ( i == 0 ) {
    i = 1;
    return true;
  } else {
    return false;
  }
}
```

(3A)

Remember the approach is:

1. Look for a row in the matrix, r, where the unmet resource needs are less than or equal to the available resources in A. If no such row exists, the system state is unsafe.
2. Assume the process from r gets all the resources it needs. Mark that process as terminated and put all its resources into A, the available pool.
3. Repeat steps 1 and 2 until either: (i) all processes are marked terminated and the initial state was safe; or (ii) no process remains whose needs can be met and the initial state is unsafe.

Looking at P_0 we cannot meet its need for resource R_0. On to P_1; its resource needs can be met from the available pool. It has $[0, 1, 1, 1]$ and needs $[2, 2, 3, 3]$. So the as-yet-unfulfilled requests are $[2, 1, 2, 2]$ while the available resources are $[6, 3, 5, 4]$. Available resources exceed the as-yet-unfulfilled requests in each resource so P_1 can run to completion. Cross out P_1; its assigned resources are added to the available vector, making the new values for A: $[6, 4, 6, 5]$

Keep this process going. P_0's requests cannot be met. P_2's needs are $[3, 4, 4, 2]$ and can be met with the new A values. So cross out P_2 and add its resources to A, producing the new A: $[10, 5, 6, 7]$

P_0's unmet needs are $[7, 5, 3, 4]$ and can now be met by the available resources. Cross out P_0 and add its assigned resources to A to get $[12, 5, 8, 8]$.

Next, P_3; A is $[13, 5, 8, 9]$.

Then P_4; A is $[14, 6, 8, 9]$.

Last, P_5; A is $[15, 6, 9, 10]$.

All processes are marked as terminated. The state is therefore **safe**.

Finally, note that it didn't matter what order we chose to run the processes; any order is okay. This solution goes in ascending order, but all roads will lead to the correct result (if followed/executed correctly).

(3B)

Anything that reduces the demands on resources or increases the available resources is always going to be safe. A deadlock can only occur if there is some shortage of resources and if things are fine now, adding more resources can't make things worse.

Anything that increases the demands on resources or reduces the system's resources is potentially unsafe. Then the question is whether the resource is in use. Removing from the available resources may lead to a deadlock in the future, but that would only happen at the time when the maximum requests for a process is increased.

1. Always can be made safely; an increase in available resources.
2. Cannot be made safely if the resource was in use; a reduction in resources.
3. Cannot be made safely; the increased resource request could potentially lead to a deadlock. This is the standard situation where we would consider running the algorithm.
4. Always can be made safely. A decrease in the demands on resources.
5. Cannot be made safely: a new process always requires some resources (even if it's just memory to load the program so it can run). Other than memory, the process may have other resource requests, but those would be considered separately (see point 3).
6. Always can be made safely. Termination will always free up some resources and cannot decrease available resources, which is okay.

(3C)

For rollback to work, we first have to take checkpoints: save the state of the process. Checkpoints can be taken periodically or before a particular operation. When a deadlock is detected, the checkpoint's saved data is restored. and the process continues from that saved state. Rollback may be attempted several times before "giving up", because it does not necessarily work the first time, but we should also limit the number of times we try it to avoid being permanently stuck in a loop.

(3D)

Recall that two phase locking means: when an attempt to acquire multiple resources is made, if not all resources are acquired, then any acquired resources are released before trying again.

Deadlock occurred amongst the philosophers when everyone picked up the left chopstick and was unable to get the right chopstick (because it was in the hand of a neighbour). With two phase locking, a philosopher who picked up the left chopstick but could not get the right chopstick puts down the left chopstick, allowing his or her neighbour to pick it up. Once a philosopher does pick up both chopsticks, he or she can eat, put down the chopsticks, and depart the table, unblocking the next one, and so on, until all philosophers get to eat.