1. Problem Sets

1.1. Semaphores

(1) 

```c
(void *)some_function( void *arg ) {
    if (arg) {
        lock.P();
        shared_data1 += 1;
        shared_data2 *= shared_data1;
        lock.V();
    }
    else {
        cout << "error" << endl;
    }
} // some_function
```

What value should `lock` be initialized to? 1

Explanation: We don’t need to protect the check of `arg` as it is local to the routine, nor do we need to protect anything in the else clause, as it accesses no data, so we only need to protect the lines shown. The initial value is 1 to allow exactly one process to initially enter the critical section.

(2) Semaphore `mutex1( 0 )`, `mutex2( 0 )`;

```c
thread1() { thread2() { thread3() {
    // do some work //do some work //do some work
    .... .... ....
    // produce data1 //do stuff with data1 //do stuff with data2
    data1 := 1; //and produce data2 cout << data2 << endl;
    mutex1.V(); mutex1.P();
    data2 := data2 * data1;
    mutex2.V();
    //do more stuff //do more stuff // do more stuff
    .... .... ....
} } }
```

(3) a) No, it isn’t necessary. There is no conflict that can occur doing the V operation, just the P
   b) Yes, there are negative consequences. One innocuous one is that it can cause extra context switches if a philosopher is putting down the forks and has the lock, it will block any philosophers trying to gain access to pick up the forks until both forks are back on
the table. Worse, it could cause a deadlock. If a philosopher is trying to grab the forks, and is blocked because one or more isn’t available, then that philosopher has the lock semaphore and the philosopher holding the forks can’t grab the lock to put the forks down.

1.2. Monitors

(1) Yes it would be OK because there is none of the items referenced in the line of code are shared between threads. The variable `other` is being declared local to the function, and `who` is an argument and therefore also local to the function, so each thread has its own copy.

(2)

```cpp
// Name: peek
int ProducerConsumer::peek(){
    int value;
    enter();
    //
    // If there aren’t any full buffers, wait for one.
    //
    if( n_full == 0 ){
        wait( full_buffers );
    }
    //
    // Retrieve the next value but don’t update indices or signal
    //
    value = buffer[ next_full ];
    leave();
    return value;
}
```
Operating Systems Test 2 Study Guide Answers

(3) Note: this returns 0 to mean no data.

```c
// Name: retrieve_data
int ProducerConsumer::retrieve_data()
{
    int value = 0; // assumes that 0 isn’t a valid value

    enter();
    //
    // If there buffers filled, go retrieve data
    //
    if( n_full > 0 ){

        //
        // Retrieve the next value, count it,
        // and advance the next_full index.
        //
        value = buffer[ next_full ];
        n_full -= 1;
        next_full = ( next_full + 1 ) % N_BUF;

        //
        // If anyone was waiting for empty buffers, let one go.
        //
        signal( empty_buffers );
    }

    leave();
    return value;
}
```

1.3. Banker’s Algorithm

(1) The system is in a safe state. Here’s why. You have 1 unit left. If you give it to P1, it can complete and release the two units. Then P2 and P3 can complete in either order. When P3 is done, it frees up enough resources for P4 to complete.

(2) The system is in an unsafe state. In order to be sure a process can complete, it will need its request of all resource types to be satisfied, so you need to look at the need for each process for all resource types and see if what is available can satisfy any process’s needs. In this example we have the following needs:

<table>
<thead>
<tr>
<th>Process</th>
<th>Type 1</th>
<th>Type 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

As we have only 1 unit of each resource type available, and no process has a need that can be satisfied, so the system is unsafe.
(3) The system is in an unsafe state. If we give the 2 units to P2, we would have the following needs:

<table>
<thead>
<tr>
<th>Process</th>
<th>Need</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>3</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
</tr>
<tr>
<td>P3</td>
<td>4</td>
</tr>
<tr>
<td>P4</td>
<td>1</td>
</tr>
<tr>
<td>P5</td>
<td>3</td>
</tr>
</tbody>
</table>

We have 1 unit left, which we can give to P4, which will release only 2 units. Of the processes remaining, no process can complete with only 2 units, so the system is unsafe.

1.4. Resource Graphs

(1) Here is one complete solution:

There are many other orders in which things could complete. But one rule is that P2 must complete before P1 completes, as it hold the all the R1 resources and P1 needs one of them.
(2) Here is one complete solution:

![Initial Graph (Step 1)](image1)

Note: request lines are shown as dashed lines

At this point no other processes can complete, so the system is deadlocked, and process P2, P3, and P5 are involved.

An alternate ordering of reducing the graph, which leads to the same result is to have P4 complete first, then P1.