Unit 7 Sample Problems - Process Synchronization I

In this exercise, we will review the concepts of Process Synchronization.

- Length: 1:15 minutes with discussion.
- Questions: Q1-Q4, Q6-Q8. (optional: Q9)

1 Background

1. [Acuña] Consider using an image editor that uses a background thread to regularly save the file being edited to ensure the user doesn't lose data. This image editor contains open, save, close, and apply filter functionality. Does this program contain any possible race condition? **Explain.** [2 points]

   Ans: [Acuña]

   Yes. There is a chance the image could auto-save, then since images are large and take time to save, the user could make a small change and click save manually. This would result in two threads saving the image at once, with the result that the contents of the file might be corrupted.

2. [Acuña] Consider a program which creates three threads and uses each of them to display "SER334" in the same console window. No synchronization occurs. Does this program contain a race condition? **Explain.** Assume that print commands are atomic. [2 points]

   Ans: [Acuña]

   No. At the end of the program, the state of the program and system will be the same no matter which order the threads execute. For example, the console window will display SER334 three times and since each thread displays the exact same text, the window will look the same. From the final point of view, it doesn't matter which thread printed which SER334 because they are equivalent.

2 The Critical-Section Problem

3. [Acuña] **Explain** how an algorithm that solves the critical-section problem would help to address the issues with the bounded buffer problem. [2 points]

   Ans: [Acuña]

   If we have a solution to the critical-section problem, that we could use it to protect a critical-section of the bounded buffer problem that encapsulates all access to shared memory. If we do this, then can be no conflict in that shared data.

4. [Acuña] **Explain** how it would be possible to have a situation where programs are making progress but do not have bounded waiting time.

   Ans: [Acuña]

   Progress only means that one program is able to do something (and will) when no process is in the critical section. This does not mean there could be another program running which never gets to the critical section, or does not enter it enough times to complete. (If a program doesn't complete, then its waiting time is unbounded.) This can also be caused by the addition of many new processes which have a higher "priority" to enter the critical section.

3 Peterson's Solution

5. [Acuña] Consider the code for Peterson's Solution:

```
//shared memory
int turn = 0;
bool flag[2] = { false, false };


//for some process i
do {
    flag[i] = true;
    turn = j;
    while (flag[j] /* && turn == j */);
    //critical section
    flag[i] = false;
    //remainder section
} while (true);
```

   Notice that part of the algorithm has been commented out. **Explain** how this changes it's functionality. Will it still solve the critical section problem? **Explain.**

   Ans: [Acuña]

   This give the program a race condition since processes no longer have to wait their term. If both processes run their first line, then both flag indices will be true and both processes will be allowed to entry (even though only one should be going on the turn). If only one process makes it to the while, then the opposite process will start, and then first process will enter as well.

6. [Acuña] **Explain** why Peterson's Solution needs to assume that loads/stores are atomic. (**Hint:** one reason is related to struct padding/alignment...)

   Ans: [Acuña]

   Because instances of the algorithm load/store data in the same turn and flag variables. This is problematic in the case that those values are not word aligned, causing the compiler to generate more than one load/store operation (meaning that store could save the first part of a new value, then a load could read the entire value, and then store would save the remaining bits of the value; potentially producing inconsistent state).

4 Synchronization Hardware

7. [Acuña] Why would it be appropriate to say that an assignment operation (a = 5;) is typically an atomic operation while an increment operation (a++;) is not? Will your answer be the same for all possible hardware? **Explain.**

   Ans: [Acuña]

   An assignment operation has only an value on the right side, so it will compile down to just an assignment without any preconditions. Since it's just a single operation, it's naturally atomic. For the increment, we need to emit code that loads, adds, and stores the value. Since we have multiple operations, it cannot be atomic.

5 Mutex Locks

8. [Lisonbee] The following code creates 4 new threads and increments and prints the contents of a global variable.

```c
#include <stdlib.h>
#include <stdio.h>

#define NUM_THREADS 4

pthread_mutex_t lock;
int data = 0;

void* runner(void* arg) {
        pthread_mutex_lock(&lock); //ACQUIRE LOCK

        data++;
        printf("Thread %d: %d\n", *((int*)arg), data);

        pthread_mutex_unlock(&lock); //RELEASE LOCK
        pthread_exit(0);
}

int main() {
        pthread_t threads[NUM_THREADS];

        pthread_mutex_init(&lock, NULL);
        for (int i = 0; i < NUM_THREADS; i++) {
                pthread_create(threads[i], NULL, runner, (i + 1));
        }
        for (int i = 0; i < NUM_THREADS; i++) {
                pthread_join(threads[i], NULL);
        }

        pthread_mutex_destroy(&lock);
        return 0;
}
```

What would happen to the output of the program if the commented lines were removed (ACQUIRE LOCK & RELEASE LOCK)? What impact do these lines have on the program's final state? **Justify** your answer.

Ans: [Lisonbee]

If these lines of code were removed then the value of data could change before a thread has had the chance to print the value. This could mean that the output could look like this:

```
Thread #: 2
Thread #: 2
Thread #: 4
Thread #: 4
```

In this instance, 'data' was incremented twice by two different threads before it was printed out, so the same value shows up twice. This happened again for the second two threads, causing the output to be 2, 2, 4, 4. The commented lines of code ensure that the value contained in 'data' is always incremented once and then printed. The order of threads that print may change from run to run, but since the data variable is incremented and printed out within the critical section of code (where the thread has the mutex lock) the value of data will always be incremented by 1 each time it's printed.

The program's final state is deterministic in terms of the data value that is printed out by all of the threads. Keeping the commented lines of code means that the output will always look like:

```
Thread #: 1
Thread #: 2
Thread #: 3
Thread #: 4
```

The '#' can change between runs because it isn't guarenteed which thread will execute the runner function first, however the data value will always be output in the order 1, 2, 3, 4.

9. [Lisonbee] Given the following partially implemented code, **implement** the calls to mutex lock and unlock in the appropriate spots in the runner function to ensure that two threads don't write to the same index in the memory array.

```
#include <stdlib.h>
#include <stdio.h>

#define NUM_THREADS 10

pthread_mutex_t lock;
int counter = 0;
int memory[NUM_THREADS];

void* runner(void* arg) {


        memory[counter] = *((int*)arg);


        printf("Wrote␣%d␣at␣index␣%d\n", *((int*)arg), counter);


        counter++;


        pthread_exit(0);


}

int main() {
        pthread_t threads[NUM_THREADS];

        pthread_mutex_init(&lock, NULL);
        for (int i = 0; i < NUM_THREADS; i++) {
                pthread_create(threads[i], NULL, runner, (i * 7));
        }
        for (int i = 0; i < NUM_THREADS; i++) {
                pthread_join(threads[i], NULL);
        }

        pthread_mutex_destroy(&lock);
        return 0;
}
```

Ans: [Lisonbee]

```
void* runner(void* arg) {
        mutex_lock(&lock);
        memory[counter] = *((int*)arg);
        printf("Wrote␣%d␣at␣index␣%d", *((int*)arg), counter);
        counter++;
        mutex_unlock(&lock);
        pthread_exit(0);
}
```