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]

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]

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]

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

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.**

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

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.**

5 Mutex Locks

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

```
#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.

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;
}
```