Roll Num: Name:

CS3523: Operating Systems 2

Quiz2 – Spring 2021

Instructions: Please give justification for all your answers

1. (a) In class we saw a solution for mutual exclusion using test_and_set() that satisfies bounded waiting. Even though this solution satisfies bounded waiting, it is not fair. Can you explain how with an example? **(3 pts)**

(b) An intuitive solution for achieving fairness is to use queue instead of array in the above implementation. What is the problem with this approach? Please explain with examples **(3 pts)**

2. Consider a system consisting of *n* processes/threads and *k* identical resources where $k<n$. Each thread of which has a unique priority number. Each thread request for a single resource at a time. If the resource is available then it is immediately allocated. If the resource is not available then the thread is made to wait. Thus in the course of time, multiple threads may wait to access the resource. Once the resource is available, the resource is given to the waiting thread with the highest priority.

Develop a monitor that achieves this. Specifically, the monitor should the following methods:

void request-resource(int thr_priority); // Here thr_priority is the priority of the invoking thread

void release-resource();

Important Note: This problem becomes trivial with ***conditional-wait*** construct of monitors mentioned in section 6.7.3 (page 281) of the 10th Edition of the Galvin book. So, please don't use this construct in your solution.
(6 pts)

3. The book has shown the implementation of monitors - hold and wait using semaphores. Can you please develop a solution for implementing monitors for hold and continue case using semaphores. **[10 pts]**

4. Some platforms provide a pair of instructions that work in concert to help build critical sections. On the MIPS architecture, for example, the load-linked and store-conditional instructions can be used in tandem to build locks and other concurrent structures. The C pseudocode for these instructions is shown below. Alpha, PowerPC, and ARM provide similar instructions.

```
// Actions of thread Ti
1: int LoadLinked(int *ptr) {
2:      return *ptr;
3: }

4: int StoreConditional(int *ptr, int value) {
5:      if (no other thread has updated *ptr since the last LoadLinked access to *ptr by thread Ti)
6:      {
7:              *ptr = value;
8:              return 1; // success!
```

```
9:      }
10:     else {
11:             return 0; // failed to update
12:     }
13: }
```

The load-linked operates much like a typical load instruction, and simply fetches a value from memory and places it in a register. The key difference comes with the store-conditional, which only succeeds (and updates the value stored at the address just load-linked from) if no intervening store by another thread to the address has taken place. In the case of success, the store-conditional returns 1 and updates the value at ptr to value; if it fails, the value at ptr is not updated and 0 is returned.

Please develop a mutex lock using this instruction. **(5 pts)**

5. The following question is on Banker's algorithm. Consider the following snapshot of a system:

Processes	**Allocation**				**Max**			
	A	**B**	**C**	**D**	**A**	**B**	**C**	**D**
P0	0	0	1	2	0	0	1	2
P1	1	0	0	0	1	7	5	0
P2	1	3	5	4	2	3	5	6
P3	0	6	3	2	0	6	5	2
P4	0	0	1	4	0	6	5	6

(a) You are given that Available = (1, 5, 2, 0). Using the banker's algorithm, show that the state is safe. **(3 pts)**

(b) You are given that Available = (1, 5, 2, 0). Consider the case that P1 requests for the resources **(0,4,2,1)**. Can this be granted immediately? **(3 pts)**

6. Suppose you wish to implement Deadlock Detection algorithm in a decentralized manner.
(a) Please explain where will you store the data-structures required by the algorithm be stored: Available, Allocation, Request, Work, Finish. **(2 pts)**

(b) How processes/threads will modify these data-structures to ensure the correct working of algorithm? Specifically explain what are the techniques you will use to protect the data-structure used by the algorithm and ensure correct synchronization. **(3 pts)**

7. (a) In the class, we discussed a solution to dining philosopher's problem using monitors. But this solution is susceptible to process starvation. Please explain how? **(2 pts)**

(b) How can the problem of starvation in dining philosopher's be solved? Please provide the pcode for this using semaphores/locks. **(5 pts)**

8. A clan of people eat their dinners from a large pan that can hold *K* servings of food. When a hungry person wants to eat, s/he helps himself from the pan with one serving, unless it is empty. If

the pan is empty, the person wakes up the **cook** and then waits until the cook has refilled the pan.

A hungry person thread executes the following unsynchronized code:

```
while (true) {
        getServingFromPan();
        eat();
}
```

Thus as shown above, a hungry person *p* will invoke this *getServingFromPan()* as long as he is hungry and there is food in the pan. In a single invocation of *getServingFromPan()*, the *p* will get one serving.

The single cook thread runs the following unsynchronzied code:

```
while (true)  {
        putServingsInPan(K);
}
```

The synchronization constraints are:

- Persons cannot invoke getServingFromPan if the pan is empty.
- The cook can invoke putServingsInPan only if the pan is empty.

Requirement: Add code for the person and the cook that satisfies the synchronization constraints. **(7 pts)**