Theory Assignment - 2

Ananya Mantravadi
CS19B1004

2)

```
monitor r {

        int available;
        condition x[k];
        queue threads;
        //priority queue storing all incoming threads according to thr_priority

        void request-resource(int thr_priority)
        {
                queue.push(thr_priority);
                        if(!available)
                                x.wait();
                        available--;
                        threads.extract_max();
                        //pop highest priority thread & assign a resource
        }

        void release-resource()
        {
                available++;
                x.signal();
        }

        initialization_code
        {
                int available = k;
        }

}
```

4) Given instructions with their pseudocodes:

```
int LoadLinked(int *ptr)
        return *ptr;
```

```
int StoreConditional(int *ptr, int value) {
        if (no other thread has updated *ptr since the last LoadLinked access to *ptr by thread Ti){
                *ptr = value;
                return 1; // success!
        }
        else
                return 0; // failed to update
}
```

A mutex-lock with the functions "acquire" and 'release" can be developed with the above instructions:

```
void acquire (lock_t *lock) {
        while (true) {
                while (LoadLinked(&lock → flag) == 1) ;
                        if (StoreConditional(&lock → flag, 1) == 1)
                                return;
        }
}
```

When a thread enters this function, it spins inside LoadLinked() till the flag which is initially 0 (indicating there is no lock on it) is set to 1. Then, it tries to acquire the lock from StoreConditional(). This is done atomically since the function will exit the while loop and return only if StoreConditional() is successful, which means there was no other thread that tried to update &lock → flag to 1.

```
void release (lock_t *lock) {
        &lock → flag = 0;
}
```

The lock on the thread is released by updating the flag value to 0.

6)

a) If we have to implement the deadlock detection algorithm in a decentralized manner, we have to store all the data structures globally, so that multiple threads can access the common data.

b) While running this algorithm, only Work and Finish are modified. We have to ensure that the other data structures Allocation, Request, and Available cannot be modified. We can either have a global lock on all the data structures, where each thread has to obtain the lock to access/update it. The drawback here will be that the whole program will slow down if one of the threads is slow. Another way is to use semaphores or mutex locks on each data structure to ensure that they are accessed/updated atomically and mutual exclusion holds to avoid race conditions. In this case, to ensure that we do not have deadlocks while accessing these data structures, we access them in a given order.

8)

```
semaphore mutex = 1;
semaphore serve = 0;
semaphore servings = K;
```

The semaphore servings keeps track of the current number of servings in the pan. The semaphore mutex is used to ensure mutual exclusion between threads in each function. Semaphore serve is used to signal/wake up the cook whenever the pan becomes empty.

```
while(true){
        void cook(){
                serve.wait();
                mutex.wait();
                putServingsInPan(K);            // add k servings to the pan here
                for(i=0;i<k;i++)
                        servings.signal();
                mutex.signal();
        }

        void person(){
                if(servings.getvalue()==0)
                        serve.signal();
                mutex.wait();
                servings.wait();
                getServingFromPan();            //reduce/eat one serving here
                eat();
                mutex.signal();
        }
}
```