Operating Systems - II
Theory Assignment - 2

2. There are n threads and k resources.

```
monitor ResourceAllocation
{
    int resources_available;
    int waiting[n];
    priority_queue<int> priority;  // Decreasing order of priorities
    condition P[n];
    initialization_code()                    // Initialising the variables
    {
        resources_available = k;
        memset(waiting, INT_MIN, n*sizeof(waiting[0]));
    }
    void request-resource(int thr_priority)
    {
        if (resources_available > 0)    // If resources are available allocate them
        {
            resources_available--;
        }
        else    // If resource is not available
        {
            int current = 0;
            // Find the first place in waiting array that is not occupied by some
            other resource
            for(int i=0;i<n;i++)
            {
                if(waiting[i]==INT_MIN)
                {
                    current = i;
                    break;
                }
            }
            priority.push(thr_priority); // Push this priority into the queue
            waiting[current] = thr_priority; // Add the priority into the waiting
            P[current].wait();        // Wait in condition variable P[current]
            // If it breaks out of wait then it uses the resource, hence decrement it
            resources_available--;
        }
    }

    void release-resource()
    {
        resources_available++; // As resource is available increment it
        if(!priority.empty())      // If there are process in priority queue
        {
            // Get the top element as it has highest priority
            int max_priority = priority.top();
            priority.pop();
            int access;
            // Find the position where it was placed in waiting, condition array
            for(int i=0;i<n;i++)
            {
                if(waiting[i]==max_priority)
                {
                    access = i;
                    break;
                }
            }
            // Reinitialise it as this can be used again
            waiting[access] = INT_MIN;
            P[access].signal();      // Signal
        }
    }
}
```
For this I used a priority queue to store the priorities, condition array and waiting array. Resources_available keeps track of the number of resources present.

Request-resource : If a process comes and finds that there are resources available then it is immediately allocated to it, else we find the first non occupied index in the waiting array and store this process's priority in it and wait on the condition variable present at this index.

Release-resource : As resource is being released we increment the resources_available variable. If there are elements in the priority queue, we use *top()* and obtain the first priority, we traverse through the waiting array to find where this priority is located and get that index. We use this index to signal the process in condition variable. If there are no elements in the priority queue we simply return

4. Mutex lock is implemented using 2 functions i.e acquire() and release(). Assuming the acquire and release functions are implemented atomically.

```
available = 1               // Initialization condition
acquire()
{
        while(1)
        {
                while(LoadLinked(available) == 0);        // Busy waiting
                if(StoreConditional(available,0) == 1)     // Loop until available is set to 0
                        return;
        }
}

release()
{
        while(StoreConditional(available,1)!= 1);          // Loop until available is set to 1
}
```

Acquire : First we use the LoadLinked() function to obtain the value of available(lock) we wait until the available is 1. Once we acquire lock we need to change it's value to 0, we do this by using the StoreConditional() function. If it was not successful in changing the value of available, we loop again because of the while(1) loop.
Release : In the release function of mutex, we need to change the value of available to 1. We use the StoreConditional() function to do this. This is placed in a while loop so that it loops until it changes value successfully.

6.

A. We can use 2d arrays or vectors to store Available, Allocation, Request, Work, Finish. We can store all the data structures globally so that they can be accessible in any function. But we have to make sure that if we are writing something to these data structures we have to use proper synchronisation techniques.
B. Race conditions will occur if there are two threads wanting to write into these data structures. Inorder to make it race free we use the following technique.

- **Using 1 lock** : Here we use only one lock, so whenever you want to modify an array you have to acquire this lock. As only one process can acquire a lock at a time, there will be no race conditions.

8.

```
int servings = K;               // Variable that keeps track of number of servings available
semaphore mutex = 1;            // Allow one person to take food at a time
semaphore doneServing = 1;  // We wait on this until cook has completed serving
semaphore cook = 0;         // Used to wake up the cook
```

Person :

```
while(true)
{
    wait(mutex)           // Allows one thread at a time
    if(serving == 0)      // If there are no servings left we wait
    {
        signal(cook);     // Waking up the chef to serve
        wait(doneServing); // Waiting until cook completes seving K portions
        servings = K;     // Updating servings
    }
    servings--;
    getServingFromPan();
    signal(mutex);
    eat();
}
```

Chef :

```
while(true)
{
        wait(cook);
        putServingsInPan(K);
        signal(doneServing);
}
```

Here I used 3 locks. The variable servings is to keep track of the number of servings available at that point of time.

Person : We make sure one person takes a serving at a time using a mutex lock. If the person finds out that there are 0 servings he signals the cook thread. Until the cook puts the servings, the person waits on doneServing lock. Once the servings are non zero, the person takes one serving from the pan using getServingFromPan() function and decrements the servings variable. After this we signal the mutex to let another person take servings from the pan.

Chef : At the beginning chef waits on the cook lock, if this lock is signaled it means that there are no servings left and the chef has to put the servings in the pan. If the chef is signalled, invoke putServingsInPan() function. Once he is done, he signals the doneServing lock, indicating that servings are added and the person can take them.