Operating Systems-II
Report-Asgn-4

2) N threads and K resources

monitor ResourceAllocation

```
{
        int resources_available;
        int waiting[n];
        condition P[n];
        priority_queue<int> priority; // Decreasing order of priorities

        init_params()  {
        memset(waiting, INT_MIN, n*sizeof(waiting[0]));
        resources_available = k;
        }

        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 for array that is not occupied by some another 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
                }
        }
}
```

I used a max priority queue to store the priorities, condition array and waiting array.

Request-resource: If a process arrives and discovers that resources are available, they are automatically assigned to it; otherwise, we select the first non-occupied index in the waiting array, store this process's priority in it, and wait on the state variable present at this index.

Release-resource: We increase the resources available variable as the resource is released. If there are items in the priority queue, we use top() to get the first priority, then we traverse the waiting array to see where this priority is stored and get the index. We use this index to signal the process in the condition variable. If there are no items in the priority queue, we clearly return.

4) We can fill the functions as follows,

```
        // lock=0 -> available, else lock is held
                        void acquire (mutex lock) {
                                while(true){
                                        while(LoadLinked(lock) == 1) ; //busy wait spin until its 0
                                        if(StoreConditional(lock,1)==1) return; //if lock is 1 then it succeeded!
                                }
                        }
```

```
void release (mutex lock){
while(!StoreConditional(lock,0)) ; // update lock value to 0 atomically
}
```

Acquire: We use the LoadLinked() function to obtain the value of the lock and wait until its value is 0 which indicates that the lock is available. Once we acquire the lock we need to change its value to 1 indicating that the lock is acquired by this thread and all others must wait. We can do this by using the StoreConditional() function which sets the lock to 1 atomically, else if it fails we loop in while(true).

Release: In the release function of mutex, we need to change the value of the lock to 0 indicating that the lock is now available and other threads can enter the CS if they wanted to. We use the StoreConditional() function to do this. We update the value of lock to 0 using StoreConditional atomically.

6) **A**. In the decentralized approach, We can use 2d arrays or vectors to store Available, Allocation, Request, Work, Finish. These all can be declared as global structures and shared to all the threads. We have to make sure that if we are writing something to these data structures we have to use proper synchronization techniques.

B. Race conditions will occur if there are two threads that want to write into these data structures. So among all the data structures, a mutex lock is also shared with these threads, before modifying the data a thread can acquire the mutex lock and update the data structure and release it. As only one process can acquire a lock at a time, there will be no race conditions.

8)

```
int servings=0
mutex_t cook=0, upd_serving=1, one_at_time=1
```

Person:

```
        while(true){
        if(serving==0)
                wait(cook); // wait for cook
        wait(one_at_time);
        getServingFromPan();
        wait(upd_serving); //update automatically
                --servings;
        signal(upd_serving);
        eat();
        signal(one_at_time);
}
```

Cook:

```
        while(true){
        if(servings==0){
                putServingsInPan(K);
                wait(upd_serving);
                serving=k;
                signal(upd_serving);
                signal(cook);
        }
}
```

Explanation:

I have used 3 mutex locks to ensure synchronization.

Person:

If there are no servings left person will wait for the cook until he fills up the servings, else one person at a time will take servings from the pan and eat it. One person at a time is ensured using the lock one_at_time.

Chef:

Chef sees that if there are zero servings, else he does nothing. If there are zero servings, he puts K servings to the pan and signals the cook lock, and hence other people can eat now.