Week 4 – Assignments

1. Explain in detail and Write a C Program for Two City Scheduling (A/P)

```
#include <bits/stdc++.h>
using namespace std;
class Solution {
  public:
  static bool cmp(vector <int> a, vector <int> b){
    return abs(a[0] - a[1]) > abs(b[0] - b[1]);
  }
  int twoCitySchedCost(vector<vector<int>>& costs) {
    int n = costs.size();
    int a = n/2;
    int b = n/2;
    sort(costs.begin(), costs.end(), cmp);
    int ans = 0;
    for(int i = 0; i < n; i++){
      if(b == 0 || (costs[i][0] <= costs[i][1] && a > 0)){
        a--;
        //cout << a << " " << costs[i][0] << endl;
        ans += costs[i][0];
      } else {
        //cout << costs[i][1] << endl;
        b--;
        ans += costs[i][1];
      }
    }
    return ans;
  }
};
main(){
  Solution ob;
  vector<vector<int>> c = {{10,20},{30,200},{400,50},{30,20}};
  cout << ob.twoCitySchedCost(c);
}
```

Input

```
[[10,20],[30,200],[400,50],[30,20]]
```

Output

```
110
```

2. Explain in detail and Write a C Program for Weighted Job Scheduling (A/P)

```
// C++ program for weighted job scheduling using Naive Recursive Method
#include <iostream>
#include <algorithm>
using namespace std;

// A job has start time, finish time and profit.
struct Job
{
	int start, finish, profit;
};

// A utility function that is used for sorting events
// according to finish time
bool jobComparator(Job s1, Job s2)
{
	return (s1.finish < s2.finish);
}

// Find the latest job (in sorted array) that doesn't
// conflict with the job[i]. If there is no compatible job,
// then it returns -1.
int latestNonConflict(Job arr[], int i)
{
	for (int j=i-1; j>=0; j--)
```

```
        {
                if (arr[j].finish <= arr[i-1].start)
                        return j;
        }
        return -1;
}

// A recursive function that returns the maximum possible
// profit from given array of jobs. The array of jobs must
// be sorted according to finish time.
int findMaxProfitRec(Job arr[], int n)
{
        // Base case
        if (n == 1) return arr[n-1].profit;

        // Find profit when current job is included
        int inclProf = arr[n-1].profit;
        int i = latestNonConflict(arr, n);
        if (i != -1)
        inclProf += findMaxProfitRec(arr, i+1);

        // Find profit when current job is excluded
        int exclProf = findMaxProfitRec(arr, n-1);

        return max(inclProf, exclProf);
}
```

```
// The main function that returns the maximum possible
// profit from given array of jobs
int findMaxProfit(Job arr[], int n)
{
        // Sort jobs according to finish time
        sort(arr, arr+n, jobComparator);

        return findMaxProfitRec(arr, n);
}

// Driver program
int main()
{
        Job arr[] = {{3, 10, 20}, {1, 2, 50}, {6, 19, 100}, {2, 100, 200}};
        int n = sizeof(arr)/sizeof(arr[0]);
        cout << "The optimal profit is " << findMaxProfit(arr, n);
        return 0;
}
```

3. Explain in detail and Write a C Program for Maximum Profit Job Scheduling (A/P)

```
#include <bits/stdc++.h>
using namespace std;
struct Data{
  int s,e,c;
  Data(int x, int y, int z){
    s= x;
```

```
    e= y;
    c = z;
  }
};
bool cmp(Data a, Data b){
  return a.e<b.e;
}
class Solution {
  public:
  int jobScheduling(vector<int>& s, vector<int>& e, vector<int>& p){
    vector<Data> j;
    int n = s.size();
    for (int i = 0; i < n; i++) {
      Data temp(s[i], e[i], p[i]);
      j.push_back(temp);
    }
    sort(j.begin(), j.end(), cmp);
    vector<int> dp(n);
    dp[0] = j[0].c;
    for (int i = 1; i < n; i++) {
      int temp = 0;
      int low = 0;
      int high = i - 1;
      while (low < high) {
        int mid = low + (high - low + 1) / 2;
        if (j[mid].e <= j[i].s)
          low = mid;
        else
          high = mid - 1;
      }
      dp[i] = j[i].c;
      if (j[low].e <= j[i].s)
        dp[i] += dp[low];
      dp[i] = max(dp[i], dp[i - 1]);
    }
    return dp[n - 1];
```

```
  }
};
main(){
  Solution ob;
  vector<int> startTime = {1,2,3,3}, endTime = {3,4,5,6}, profit =
  {500,100,400,700};
  cout << (ob.jobScheduling(startTime, endTime, profit));
}
```

4. Explain in detail and Write a C Program for QOS Traffic Scheduling (A/P)

QoS traffic scheduling is a scheduling methodology of network traffic based upon QoS (Quality of Service). Here, the frames or packets are mapped to internal forwarding queues based on its QoS information, which are then services according to a queuing scheme.

Typically, multiple queues are present each with different priority levels. The scheduler decides the type of treatment to be given to the traffic in each queue. When traffic is available, the scheduler maps it to the appropriate queue. For example, video and voice traffic are kept is queued with higher priority than background traffic.

Notable QoS Traffic Scheduling Methods

- **Weighted Round Robin (WRR)** – In this method, frames or packets of all the queues in the scheduler are serviced in each cycle. Priority among the different queues is maintained by forwarding a specific number of frames in each queue per cycle in a rotational manner. For example in a system with four queues $Q0$, $Q1$, $Q2$, $Q3$, 5 frames of $Q0$, 3 frames of $Q1$, 1 frame each of $Q2$ and $Q3$ may be sent in each cycle. This assigns the highest priority and consequently better traffic flow to $Q0$ while not starving low priority traffic infinitely.
- **Strict Priority (SP)** – This method imparts the highest service to high priority traffic. Here, the queuing mechanism forwards as many frames as possible in a higher priority frame before moving to the queue with the next priority level. For example in a system with four queues $Q0$, $Q1$, $Q2$, $Q3$, with decreasing priority, strict priority method with send as many frames as permissible of $Q0$ before moving on to $Q1$.

- **Combination of WRR and SP** – This queuing method is configurable in nature and it combines both WRR and SP. In this method, strict priority is given to time-sensitive or real-time traffic like voice and video, while WRR is adopted for other traffics.

5. Explain in detail and Write a C Program for Highest Response Ratio Next (HRNN) optimal scheduling algorithm (A/P)

```
// C program for Highest Response Ratio Next (HRRN) Scheduling
#include <stdio.h>

// Defining process details
struct process {
	char name;
	int at, bt, ct, wt, tt;
	int completed;
	float ntt;
} p[10];

int n;

// Sorting Processes by Arrival Time
void sortByArrival()
{
	struct process temp;
	int i, j;

	// Selection Sort applied
```

```
        for (i = 0; i < n - 1; i++) {
                for (j = i + 1; j < n; j++) {

                        // Check for lesser arrival time
                        if (p[i].at > p[j].at) {

                                // Swap earlier process to front
                                temp = p[i];
                                p[i] = p[j];
                                p[j] = temp;
                        }
                }
        }
}

void main()
{
        int i, j, t, sum_bt = 0;
        char c;
        float avgwt = 0, avgtt = 0;
        n = 5;

        // predefined arrival times
        int arriv[] = { 0, 2, 4, 6, 8 };

        // predefined burst times
```

```
int burst[] = { 3, 6, 4, 5, 2 };

// Initializing the structure variables
for (i = 0, c = 'A'; i < n; i++, c++) {
        p[i].name = c;
        p[i].at = arriv[i];
        p[i].bt = burst[i];

        // Variable for Completion status
        // Pending = 0
        // Completed = 1
        p[i].completed = 0;

        // Variable for sum of all Burst Times
        sum_bt += p[i].bt;
}

// Sorting the structure by arrival times
sortByArrival();
printf("\nName\tArrival Time\tBurst Time\tWaiting Time");
printf("\tTurnAround Time\t Normalized TT");
for (t = p[0].at; t < sum_bt;) {

        // Set lower limit to response ratio
        float hrr = -9999;
```

```
// Response Ratio Variable
float temp;

// Variable to store next process selected
int loc;
for (i = 0; i < n; i++) {

        // Checking if process has arrived and is Incomplete
        if (p[i].at <= t && p[i].completed != 1) {

                // Calculating Response Ratio
                temp = (p[i].bt + (t - p[i].at)) / p[i].bt;

                // Checking for Highest Response Ratio
                if (hrr < temp) {

                        // Storing Response Ratio
                        hrr = temp;

                        // Storing Location
                        loc = i;
                }
        }
}

// Updating time value
```

```
            t += p[loc].bt;

            // Calculation of waiting time
            p[loc].wt = t - p[loc].at - p[loc].bt;

            // Calculation of Turn Around Time
            p[loc].tt = t - p[loc].at;

            // Sum Turn Around Time for average
            avgtt += p[loc].tt;

            // Calculation of Normalized Turn Around Time
            p[loc].ntt = ((float)p[loc].tt / p[loc].bt);

            // Updating Completion Status
            p[loc].completed = 1;

            // Sum Waiting Time for average
            avgwt += p[loc].wt;
            printf("\n%c\t\t%d\t\t", p[loc].name, p[loc].at);
            printf("%d\t\t%d\t\t", p[loc].bt, p[loc].wt);
            printf("%d\t\t%f", p[loc].tt, p[loc].ntt);
        }
        printf("\nAverage waiting time:%f\n", avgwt / n);
        printf("Average Turn Around time:%f\n", avgtt / n);
}
```

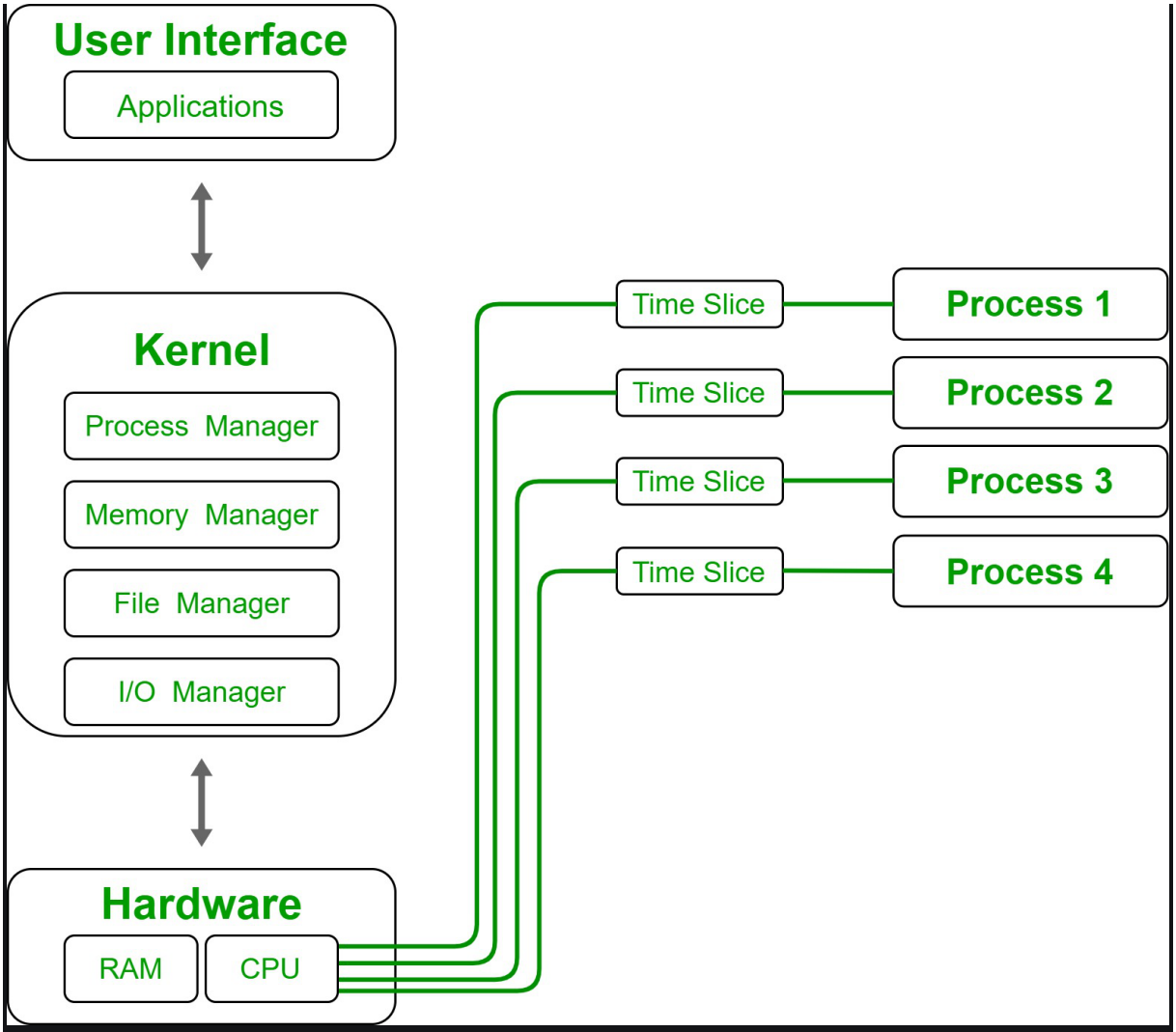
6. Explain in detail the role of Time Slicer in CPU Scheduling.

CPUs kernel doesn’t simply distribute the entirety of our PCs’ resources to single process or service. CPU is continuously running many processes that are essential for it to operate, so our kernel needs to manage these processes without moment’s delay.

When program needs to run, process must be created for it. This process needs to have important resources like RAM and CPU. The kernel schedules time periods for CPU to perform commands and instructions in process. Be that as it may, there’s just single CPU and numerous processes.

How does CPU outstand to execute different processes without moment’s delay? It does it by executing processes one by one, individually by time slice. A time slice is short time frame that gets assigned to process for CPU execution.

Time slice :
It is timeframe for which process is allotted to run in preemptive multitasking CPU. The scheduler runs each process every single time-slice. The period of each time slice can be very significant and crucial to balance CPUs performance and responsiveness.
If time slice is quite short, scheduler will take more processing time. In contrast, if the time slice is too long, scheduler will again take more processing time.



When process is allotted to CPU, clock timer is set corresponding to time slice.

- If the process finishes its burst before time slice, CPU simply swaps it out like conventional FCFS calculation.
- If the time slice goes off first, CPU shifts it out to back of ongoing queue.

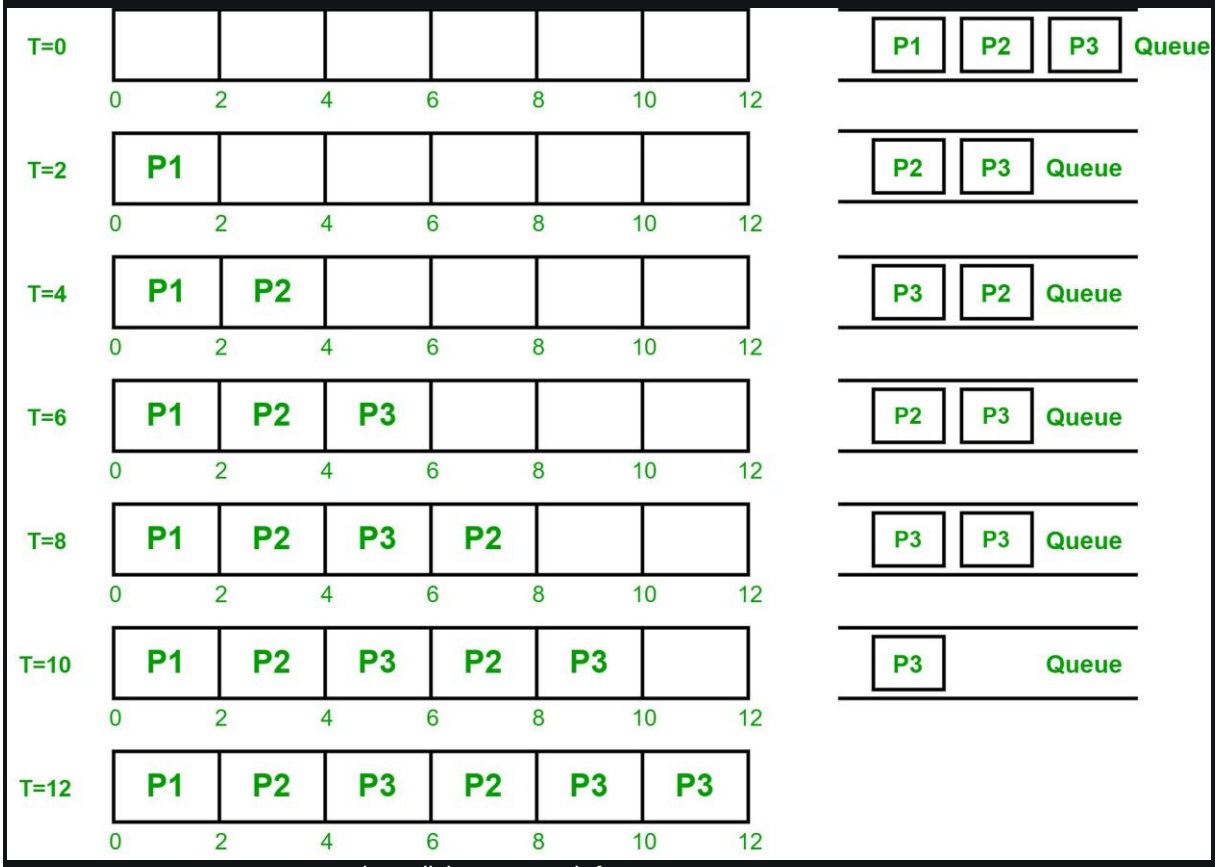
The ongoing queue is managed like circular queue, so, after all processes are executed once, scheduler executes first process again and then second and so forth.

Example –

Process Queue	Required burst time by process(ms)
P1	1
P2	4

P3 5

We have three processes(P1, P2, P3) with their corresponding burst times(1ms, 4ms, 5ms). A rule of thumb is that 80% of CPU bursts should be smaller than the time quantum. Considering time slice of 2ms.
Here's how CPU manages it by time slicing.



time-slicing approach for process management

Advantages :

- Fair allocation of CPU resources.
- It deals all process with equal priority.
- Easily implementable on the system.
- Context switching method used to save states of preempted processes
- gives best performance in terms of average processing time.

Disadvantages :

- If the slicing time is short, processor output will be delayed.
- It spends time on context switching.
- Performance depends heavily on time quantum.
- Priorities can't be fixed for processes.
- No priority to more important tasks.
- Finding an appropriate time quantum is quite difficult.