1. MULTILEVEL CPU SCHEDULING:

```
#include <stdio.h>

#define MAX_QUEUE_SIZE 100

// Structure to represent a process
typedefstruct {
intprocessID;
intarrivalTime;
intburstTime;
int priority; // 0 for system process, 1 for user process
} Process;

// Function to execute a process
voidexecuteProcess(Process process) {
printf("Executing Process %d\n", process.processID);
    // Simulating the execution time of the process
for (inti = 1; i<= process.burstTime; i++) {
printf("Process %d: %d/%d\n", process.processID, i, process.burstTime);
    }
printf("Process %d executed\n", process.processID);
}

// Function to perform FCFS scheduling for a queue of processes
voidscheduleFCFS(Process queue[], int size) {
for (inti = 0; i< size; i++) {
executeProcess(queue[i]);
    }
}

int main() {
intnumProcesses;
    Process processes[MAX_QUEUE_SIZE];

    // Reading the number of processes
printf("Enter the number of processes: ");
scanf("%d", &numProcesses);
```

```
    // Reading process details
for (inti = 0; i<numProcesses; i++) {
printf("Process %d:\n", i + 1);
printf("Arrival Time: ");
scanf("%d", &processes[i].arrivalTime);
printf("Burst Time: ");
scanf("%d", &processes[i].burstTime);
printf("System(0)/User(1): ");
scanf("%d", &processes[i].priority);
processes[i].processID = i + 1;
    }

    // Separate system and user processes into different queues
    Process systemQueue[MAX_QUEUE_SIZE];
intsystemQueueSize = 0;
    Process userQueue[MAX_QUEUE_SIZE];
intuserQueueSize = 0;

for (inti = 0; i<numProcesses; i++) {
if (processes[i].priority == 0) {
systemQueue[systemQueueSize++] = processes[i];
        } else {
userQueue[userQueueSize++] = processes[i];
        }
    }

    // Execute system queue processes first
printf("System Queue:\n");
scheduleFCFS(systemQueue, systemQueueSize);

    // Execute user queue processes
printf("User Queue:\n");
scheduleFCFS(userQueue, userQueueSize);

return 0;
}
```

OUTPUT:

"C:\Users\Admin\Desktop\4th Sem\Lab\OS LAB\Multilevel Scheduling.exe"

```
Enter the number of processes: 6
Process 1:
Arrival Time: 0
Burst Time: 3
System(0)/User(1): 0
Process 2:
Arrival Time: 2
Burst Time: 2
System(0)/User(1): 0
Process 3:
Arrival Time: 4
Burst Time: 4
System(0)/User(1): 1
Process 4:
Arrival Time: 4
Burst Time: 2
System(0)/User(1): 1
Process 5:
Arrival Time: 8
Burst Time: 2
System(0)/User(1): 0
Process 6:
Arrival Time: 10
Burst Time: 3
System(0)/User(1): 1
System Queue:
Executing Process 1
Process 1: 1/3
Process 1: 2/3
Process 1: 3/3
Process 1 executed
Executing Process 2
Process 2: 1/2
Process 2: 2/2
Process 2 executed
Executing Process 5
Process 5: 1/2
Process 5: 2/2
Process 5 executed
User Queue:
Executing Process 3
Process 3: 1/4
Process 3: 2/4
Process 3: 3/4
Process 3: 4/4
Process 3 executed
Executing Process 4
Process 4: 1/2
Process 4: 2/2
Process 4 executed
Executing Process 6
Process 6: 1/3
Process 6: 2/3
Process 6: 3/3
Process 6 executed

Process returned 0 (0x0)   execution time : 85.650 s
Press any key to continue.
```

2. RATE MONOMOTIC SCHEDULING:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdbool.h>

#define MAX_PROCESS 10

intnum_of_process = 3, count, remain, time_quantum;
intexecution_time[MAX_PROCESS], period[MAX_PROCESS],
remain_time[MAX_PROCESS], deadline[MAX_PROCESS],
remain_deadline[MAX_PROCESS];
intburst_time[MAX_PROCESS], wait_time[MAX_PROCESS],
completion_time[MAX_PROCESS], arrival_time[MAX_PROCESS];

// collecting details of processes
voidget_process_info(intselected_algo)
{
printf("Enter total number of processes (maximum %d): ",
MAX_PROCESS);
scanf("%d", &num_of_process);
if (num_of_process< 1)
   {
printf("Do you really want to schedule %d processes? -_-",
num_of_process);
exit(0);
   }
if (selected_algo == 2)
   {
printf("\nEnter Time Quantum: ");
scanf("%d", &time_quantum);
if (time_quantum< 1)
      {
printf("Invalid Input: Time quantum should be greater than 0\n");
```

```
exit(0);
       }
   }

for (inti = 0; i<num_of_process; i++)
   {
printf("\nProcess %d:\n", i + 1);
if (selected_algo == 1)
       {
printf("==> Burst time: ");
scanf("%d", &burst_time[i]);
       }
else if (selected_algo == 2)
       {
printf("=> Arrival Time: ");
scanf("%d", &arrival_time[i]);
printf("=> Burst Time: ");
scanf("%d", &burst_time[i]);
remain_time[i] = burst_time[i];
       }
else if (selected_algo> 2)
       {
printf("==> Execution time: ");
scanf("%d", &execution_time[i]);
remain_time[i] = execution_time[i];
if (selected_algo == 4)
          {
printf("==> Deadline: ");
scanf("%d", &deadline[i]);
          }
else
          {
printf("==> Period: ");
scanf("%d", &period[i]);
          }
       }
   }
```

```
}

// get maximum of three numbers
int max(int a, int b, int c)
{
int max;
if (a >= b && a >= c)
max = a;
else if (b >= a && b >= c)
max = b;
else if (c >= a && c >= b)
max = c;
return max;
}

// calculating the observation time for scheduling timeline
intget_observation_time(intselected_algo)
{
if (selected_algo< 3)
   {
int sum = 0;
for (inti = 0; i<num_of_process; i++)
       {
sum += burst_time[i];
       }
return sum;
   }
else if (selected_algo == 3)
   {
return max(period[0], period[1], period[2]);
   }
else if (selected_algo == 4)
   {
return max(deadline[0], deadline[1], deadline[2]);
   }
}
```

```
// print scheduling sequence
voidprint_schedule(intprocess_list[], int cycles)
{
printf("\nScheduling:\n\n");
printf("Time: ");
for (inti = 0; i< cycles; i++)
   {
if (i< 10)
printf("| 0%d ", i);
else
printf("| %d ", i);
   }
printf("|\n");

for (inti = 0; i<num_of_process; i++)
   {
printf("P[%d]: ", i + 1);
for (int j = 0; j < cycles; j++)
       {
if (process_list[j] == i + 1)
printf("|####");
else
printf("|    ");
       }
printf("|\n");
   }
}

voidrate_monotonic(int time)
{
intprocess_list[100] = {0}, min = 999, next_process = 0;
float utilization = 0;
for (inti = 0; i<num_of_process; i++)
   {
utilization += (1.0 * execution_time[i]) / period[i];
   }
int n = num_of_process;
```

```
if (utilization > n * (pow(2, 1.0 / n) - 1))
   {
printf("\nGiven problem is not schedulable under the said scheduling
algorithm.\n");
exit(0);
   }



for (inti = 0; i< time; i++)
   {
min = 1000;
for (int j = 0; j <num_of_process; j++)
      {
if (remain_time[j] > 0)
         {
if (min > period[j])
            {
min = period[j];
next_process = j;
            }
         }
      }

if (remain_time[next_process] > 0)
      {
process_list[i] = next_process + 1; // +1 for catering 0 array index.
remain_time[next_process] -= 1;
      }

for (int k = 0; k <num_of_process; k++)
      {
if ((i + 1) % period[k] == 0)
         {
remain_time[k] = execution_time[k];
next_process = k;
         }
      }
```

```
    }
print_schedule(process_list, time);
}



int main(intargc, char *argv[])
{
int option = 0;

printf("3. Rate Monotonic Scheduling\n");

printf("Select > ");
scanf("%d", &option);
printf("----------------------------\n");

get_process_info(option); // collecting processes detail
intobservation_time = get_observation_time(option);


if (option == 3)
rate_monotonic(observation_time);
return 0;
}
```

OUTPUT:

```
"C:\Users\Admin\Desktop\4th Sem\Lab\OS LAB\Rate Monotonic-1.exe"
3. Rate Monotonic Scheduling
Select > 3
-----------------------------
Enter total number of processes (maximum 10): 3

Process 1:
==> Execution time: 3
==> Period: 20

Process 2:
==> Execution time: 2
==> Period: 5

Process 3:
==> Execution time: 2
==> Period: 10

Scheduling:

Time: | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
P[1]: |    |    |    |    |####|    |    |####|####|    |    |    |    |    |    |    |    |    |    |    |
P[2]: |####|####|    |    |    |####|####|    |    |    |####|####|    |    |    |####|####|    |    |    |
P[3]: |    |    |####|####|    |    |    |    |    |    |    |    |####|####|    |    |    |    |    |    |

Process returned 0 (0x0)   execution time : 68.961 s
Press any key to continue.
```

3. EARLIEST DEADLINE FIRST

```
#include <stdio.h>
#define arrival          0
#define execution        1
#define deadline         2
#define period           3
#define abs_arrival      4
#define execution_copy   5
#define abs_deadline     6


typedefstruct
{
	int T[7],instance,alive;
```

```
}task;

#define IDLE_TASK_ID 1023
#define ALL 1
#define CURRENT 0

voidget_tasks(task *t1,int n);
inthyperperiod_calc(task *t1,int n);
floatcpu_util(task *t1,int n);
intgcd(int a, int b);
int lcm(int *a, int n);
intsp_interrupt(task *t1,int tmr,int n);
int min(task *t1,int n,int p);
voidupdate_abs_arrival(task *t1,int n,intk,int all);
voidupdate_abs_deadline(task *t1,int n,int all);
voidcopy_execution_time(task *t1,int n,int all);


int timer = 0;

int main()
{
	task *t;
	int n, hyper_period, active_task_id;
	floatcpu_utilization;
	printf("Enter number of tasks\n");
```

```
scanf("%d", &n);
t = malloc(n * sizeof(task));
get_tasks(t, n);
cpu_utilization = cpu_util(t, n);
printf("CPU Utilization %f\n", cpu_utilization);

if (cpu_utilization< 1)
        printf("Tasks can be scheduled\n");
else
        printf("Schedule is not feasible\n");

hyper_period = hyperperiod_calc(t, n);
copy_execution_time(t, n, ALL);
update_abs_arrival(t, n, 0, ALL);
update_abs_deadline(t, n, ALL);

while (timer <= hyper_period)
{

        if (sp_interrupt(t, timer, n))
        {
                active_task_id = min(t, n, abs_deadline);
        }

        if (active_task_id == IDLE_TASK_ID)
        {
```

```
            printf("%d  Idle\n", timer);
        }

        if (active_task_id != IDLE_TASK_ID)
        {

            if (t[active_task_id].T[execution_copy] != 0)
            {
                t[active_task_id].T[execution_copy]--;
                printf("%d  Task %d\n", timer, active_task_id + 1);
            }

            if (t[active_task_id].T[execution_copy] == 0)
            {
                t[active_task_id].instance++;
                t[active_task_id].alive = 0;
                copy_execution_time(t, active_task_id, CURRENT);
                update_abs_arrival(t, active_task_id,
t[active_task_id].instance, CURRENT);
                update_abs_deadline(t, active_task_id, CURRENT);
                active_task_id = min(t, n, abs_deadline);
            }
        }
        ++timer;
    }
    free(t);
    return 0;
```

```
}
voidget_tasks(task *t1, int n)
{
	inti = 0;
	while (i< n)
	{
		printf("Enter Task %d parameters\n", i + 1);
		printf("Arrival time: ");
		scanf("%d", &t1->T[arrival]);
		printf("Execution time: ");
		scanf("%d", &t1->T[execution]);
		printf("Deadline time: ");
		scanf("%d", &t1->T[deadline]);
		printf("Period: ");
		scanf("%d", &t1->T[period]);
		t1->T[abs_arrival] = 0;
		t1->T[execution_copy] = 0;
		t1->T[abs_deadline] = 0;
		t1->instance = 0;
		t1->alive = 0;
		t1++;
		i++;
	}
}

inthyperperiod_calc(task *t1, int n)
```

```
{
        inti = 0, ht, a[10];
        while (i< n)

        {
                a[i] = t1->T[period];
                t1++;
                i++;
        }
        ht = lcm(a, n);

        returnht;
}

intgcd(int a, int b)
{
        if (b == 0)
                return a;
        else
                returngcd(b, a % b);
}

int lcm(int *a, int n)
{
        int res = 1, i;
        for (i = 0; i< n; i++)
```

```
	{
		res = res * a[i] / gcd(res, a[i]);
	}
	return res;
}

intsp_interrupt(task *t1, inttmr, int n)
{
	inti = 0, n1 = 0, a = 0;
	task *t1_copy;
	t1_copy = t1;
	while (i< n)
	{
		if (tmr == t1->T[abs_arrival])
		{
			t1->alive = 1;
			a++;
		}
		t1++;
		i++;
	}

	t1 = t1_copy;
	i = 0;

	while (i< n)
```

```
	{
		if (t1->alive == 0)
			n1++;
		t1++;
		i++;
	}

	if (n1 == n || a != 0)
	{
		return 1;
	}

	return 0;
}

voidupdate_abs_deadline(task *t1, int n, int all)
{
	inti = 0;
	if (all)
	{
		while (i< n)
		{
			t1->T[abs_deadline] = t1->T[deadline] + t1->T[abs_arrival];
			t1++;
			i++;
		}
```

```
        }
        else
        {
                t1 += n;
                t1->T[abs_deadline] = t1->T[deadline] + t1->T[abs_arrival];
        }
}

voidupdate_abs_arrival(task *t1, int n, int k, int all)
{
        inti = 0;
        if (all)
        {
                while (i< n)
                {
                        t1->T[abs_arrival] = t1->T[arrival] + k * (t1->T[period]);
                        t1++;
                        i++;
                }
        }
        else
        {
                t1 += n;
                t1->T[abs_arrival] = t1->T[arrival] + k * (t1->T[period]);
        }
}
```

```
voidcopy_execution_time(task *t1, int n, int all)
{
	inti = 0;
	if (all)
	{
		while (i< n)
		{
			t1->T[execution_copy] = t1->T[execution];
			t1++;
			i++;
		}
	}
	else
	{
		t1 += n;
		t1->T[execution_copy] = t1->T[execution];
	}
}

int min(task *t1, int n, int p)
{
	inti = 0, min = 0x7FFF, task_id = IDLE_TASK_ID;
	while (i< n)
	{
		if (min > t1->T[p] && t1->alive == 1)
```

```
        {
            min = t1->T[p];
            task_id = i;
        }
        t1++;
        i++;
    }
    returntask_id;
}

floatcpu_util(task *t1, int n)
{
    inti = 0;
    float cu = 0;
    while (i< n)
    {
        cu = cu + (float)t1->T[execution] / (float)t1->T[deadline];
        t1++;
        i++;
    }
    return cu;
}
```

OUTPUT:

"C:\Users\Admin\Desktop\4th Sem\Lab\OS LAB\EDF-1.exe"

```
Enter number of tasks
3
Enter Task 1 parameters
Arrival time: 0
Execution time: 3
Deadline time: 7
Period: 20
Enter Task 2 parameters
Arrival time: 0
Execution time: 2
Deadline time: 4
Period: 5
Enter Task 3 parameters
Arrival time: 0
Execution time: 2
Deadline time: 8
Period: 10
CPU Utilization 1.178571
Schedule is not feasible
0  Task 2
1  Task 2
2  Task 1
3  Task 1
4  Task 1
5  Task 3
6  Task 3
7  Task 2
8  Task 2
9  Idle
10  Task 2
11  Task 2
12  Task 3
13  Task 3
14  Idle
15  Task 2
16  Task 2
17  Idle
18  Idle
19  Idle
```