

LAB1:

```

#include <stdio.h>
#include <limits.h>

void findWaitingTimeFCFS(int processes[], int n, int bt[], int wt[], int at[], int ct[]) {
    for (int i = 0; i < n; i++) {
        wt[i] = ct[i] - at[i] - bt[i];
    }
}

void findWaitingTimeSJFFree preemptive(int processes[], int n, int bt[], int wt[], int at[], int ct[]) {
    int rt[n];
    for (int i = 0; i < n; i++)
        rt[i] = bt[i];
    int complete = 0, t = 0, minm = INT_MAX;
    int shortest = 0, finish_time;
    while (complete != n) {
        for (int j = 0; j < n; j++) {
            if ((at[j] <= t) && (rt[j] < minm) && (rt[j] > 0)) {
                minm = rt[j];
                shortest = j;
            }
        }
        rt[shortest]--;
        minm = rt[shortest];
        if (minm == 0)
            minm = INT_MAX;
        if (rt[shortest] == 0) {
            complete++;
            finish_time = t + 1;
            wt[shortest] = finish_time - bt[shortest] - at[shortest];
            if (wt[shortest] < 0)
                wt[shortest] = 0;
            ct[shortest] = finish_time;
        }
        t++;
    }
}

void findWaitingTimeSJFNonPreemptive(int processes[], int n, int bt[], int wt[], int at[], int ct[]) {
    int rt[n];
    for (int i = 0; i < n; i++)
        rt[i] = bt[i];

    int complete = 0, t = 0, minm = INT_MAX;
    int shortest = 0, finish_time;
    while (complete != n) {
        for (int j = 0; j < n; j++) {
            if (at[j] <= t) {
                if (rt[j] < minm) {
                    minm = rt[j];
                    shortest = j;
                }
            }
        }
        rt[shortest]--;
        minm = rt[shortest];
        if (minm == 0)
            minm = INT_MAX;
        if (rt[shortest] == 0) {
            complete++;
            finish_time = t + 1;
            wt[shortest] = finish_time - bt[shortest] - at[shortest];
            if (wt[shortest] < 0)
                wt[shortest] = 0;
            ct[shortest] = finish_time;
        }
        t++;
    }
}

```

```

        minm = rt[i];
        shortest = i;
    }
    t += rt[shortest];
    finish_time = t;
    wt[shortest] = finish_time - bt[shortest] - at[shortest];
    if (wt[shortest] < 0)
        wt[shortest] = 0;
    rt[shortest] = INT_MAX;
    complete++;
    ct[shortest] = finish_time;
    minm = INT_MAX;
}

void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[], int ct[], int at[]) {
    for (int i = 0; i < n; i++)
        tat[i] = ct[i] - at[i];
}

void findAverageTimeFCFS(int processes[], int n, int bt[], int at[], int ct[]) {
    int wt[n], tat[n];
    int total_wt = 0, total_tat = 0;

    findWaitingTimeFCFS(processes, n, bt, wt, at, ct);
    findTurnAroundTime(processes, n, bt, wt, tat, ct, at);

    printf("FCFS Scheduling\n");
    printf("Processes Arrival time Burst time Waiting time Turn around time Completion time\n");

    for (int i = 0; i < n; i++) {
        total_wt += wt[i];
        total_tat += tat[i];
        printf("%d ", processes[i]);
        printf("%d ", at[i]);
        printf("%d ", bt[i]);
        printf("%d", wt[i]);
        printf("%d", tat[i]);
        printf("%d\n", ct[i]);
    }

    float avg_wt = (float)total_wt / n;
    float avg_tat = (float)total_tat / n;
    printf("Average waiting time = %f\n", avg_wt);
}

```

```

    printf("Average turn around time = %f\n", avg_tat);
}

void findAverageTimeSJFNonPreemptive(int processes[], int n, int bt[], int at[], int ct[]) {
    int wt[n], tat[n];
    int total_wt = 0, total_tat = 0;

    findWaitingTimeSJFNonPreemptive(processes, n, bt, wt, at, ct);
    findTurnAroundTime(processes, n, bt, wt, tat, ct, at);

    printf("\nSJF (Non-preemptive) Scheduling\n");
    printf("Processes Arrival time Burst time Waiting time Turn around time Completion time\n");

    for (int i = 0; i < n; i++) {
        total_wt += wt[i];
        total_tat += tat[i];
        printf("%d ", processes[i]);
        printf("%d ", at[i]);
        printf("%d ", bt[i]);
        printf("%d", wt[i]);
        printf("%d", tat[i]);
        printf("%d\n", ct[i]);
    }

    float avg_tat = (float)total_tat / n;
    printf("Average waiting time = %f\n", avg_wt);
    printf("Average turn around time = %f\n", avg_tat);
}

void findAverageTimeSJFFre preemptive(int processes[], int n, int bt[], int at[], int ct[]) {
    int wt[n], tat[n];
    int total_wt = 0, total_tat = 0;

    findWaitingTimeSJFFre preemptive(processes, n, bt, wt, at, ct);
    findTurnAroundTime(processes, n, bt, wt, tat, ct, at);

    printf("\nSJF (Preemptive) Scheduling\n");
    printf("Processes Arrival time Burst time Waiting time Turn around time Completion time\n");

    for (int i = 0; i < n; i++) {
        total_wt += wt[i];
        total_tat += tat[i];
        printf("%d ", processes[i]);
        printf("%d ", at[i]);
        printf("%d ", bt[i]);
        printf("%d", wt[i]);
        printf("%d", tat[i]);
    }
}

```

```

    float avg_lat = (float)total_tat / n;
    printf("Average waiting time = %f\n", avg_wt);
    printf("Average turn around time = %f\n", avg_tat);
}

int main() {
    int processes[10], burst_time[10], arrival_time[10], completion_time[10];
    int n;

    printf("Enter the number of processes: ");
    scanf("%d", &n);

    printf("Enter arrival time and burst time for each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Arrival time of process[%d]: ", i + 1);
        scanf("%d", &arrival_time[i]);
        printf("Burst time of process[%d]: ", i + 1);
        scanf("%d", &burst_time[i]);
        processes[i] = i + 1;
    }

    completion_time[0] = arrival_time[0] + burst_time[0];
    for (int i = 1; i < n; i++) {
        if (arrival_time[i] > completion_time[i - 1]) {
            completion_time[i] = arrival_time[i] + burst_time[i];
        } else {
            completion_time[i] = completion_time[i - 1] + burst_time[i];
        }
    }

    findAverageTimeFCFS(processes, n, burst_time, arrival_time, completion_time);
    findAverageTimeSJFNonPreemptive(processes, n, burst_time, arrival_time, completion_time);
    findAverageTimeSJFPreemptive(processes, n, burst_time, arrival_time, completion_time);

    return 0;
}

```

## OUTPUT:

```

Enter the number of processes: 3
Enter arrival time and burst time for each process:
Arrival time of process[1]: 0
Burst time of process[1]: 2
Arrival time of process[2]: 1
Burst time of process[2]: 4
Arrival time of process[3]: 2
Burst time of process[3]: 3
FCFS Scheduling
Processes Arrival time Burst time Waiting time Turn around time Completion time
1 0 2 0 2 2
2 1 4 1 5 6
3 2 3 4 7 9
Average waiting time = 1.666667
Average turn around time = 4.666667

SJF (Non-preemptive) Scheduling
Processes Arrival time Burst time Waiting time Turn around time Completion time
1 0 2 0 2 2
2 1 4 4 8 9
3 2 3 0 3 5
Average waiting time = 1.333333
Average turn around time = 4.333333

SJF (Preemptive) Scheduling
Processes Arrival time Burst time Waiting time Turn around time Completion time
1 0 2 0 2 2
2 1 4 4 8 9
3 2 3 0 3 5
Average waiting time = 1.333333
Average turn around time = 4.333333

```

7/1/21

### LAB PROGRAM - D)

Page No. \_\_\_\_\_  
Date \_\_\_\_\_

Write a C program to simulate the following user prompt like, CPU scheduling algorithm to find turnaround time and waiting time  
→ FCFS  
→ SJFC pre-emptive and non-pre-emptive

function estArr[n]:

int i;

and function SJFC different priorities: short, SJFC, SJF and SJF, Priority, SJF etc.

for (i=0; i < n; i++)

    estArr[i] = (tArr[i] - at[i]) / bArr[i];

and SJFC uses SJF if preemptive (not pre-emptive), SJF, SJF, SJF, SJF

int tArr[n], at[n], bArr[n], Priorities[n];

int i, j, k;

for (i=0; i < n; i++)

    tArr[i] = bArr[i] \* j;

int minTime=0, t=0, minTime=0, minTime=0, shortWait=0, finishTime=0;

while (completions != n) {

    for (i=0; i < n; i++) {

        if ((at[i] <= t) && (estArr[i] <= minTime || estArr[i] >= 0)) {

            minTime = estArr[i]; shortWait = i; }

        if (shortWait == i) shortWait = j; }

    if (minTime == 0) {

        minTime = 1; minTime = 0;

        if (at[shortWait] == 0) {

            at[shortWait] = 1;

<pre>     float_t tmax=t;     wt[shortest] = finish_time - bt[shortest] - at[shortest];     if (wt[shortest] &lt; 0)         wt[shortest] = 0;     ct[shortest] = finish_time;     hpp++; ss     void fsh() {         Pm-&gt;SP-&gt;shortest();         int min, max, i;         min = Pm-&gt;SP-&gt;min();         max = Pm-&gt;SP-&gt;max();         for (i=min; i&lt;=max; i++)             if (at[i]==0) {                 if (i&gt;=t)                     if ((i+ts)&lt;min    (i+ts)&gt;max) {                         min = i+ts;                         max = i+ts;                         shortest = i;                     }                 t += ts;             }         tmax = t;         wt[shortest] = finish_time - bt[shortest] - at[shortest];         if (wt[shortest]&lt;0)             wt[shortest] = 0;         ct[shortest] = finish_time;         min = Pm-&gt;SP-&gt;min();         max = Pm-&gt;SP-&gt;max();         for (i=min; i&lt;=max; i++)             if (at[i]==0) {                 if (i&gt;=t)                     if ((i+ts)&lt;min    (i+ts)&gt;max) {                         min = i+ts;                         max = i+ts;                         shortest = i;                     }                 t += ts;             }         tmax = t;     } </pre>	<pre>     void fsh() {         Pm-&gt;SP-&gt;shortest();         int min, max, i;         min = Pm-&gt;SP-&gt;min();         max = Pm-&gt;SP-&gt;max();         for (i=min; i&lt;=max; i++)             if (at[i]==0) {                 if (i&gt;=t)                     if ((i+ts)&lt;min    (i+ts)&gt;max) {                         min = i+ts;                         max = i+ts;                         shortest = i;                     }                 t += ts;             }         tmax = t;     } </pre>
--	--



```

    processes[i].at[i], bt[i], at[i], totalCid, wt[i]);
float avg_wt = (float) total_wt / n;
float avg_tat = (float) total_tat / n;
printf("Avg -> %f, avg_wt = %f\n", avg_tat);
printf("Avg_wt = %f/n, avg_wt = %f\n", avg_wt);
}

void FindAverageTimeJFPremptive(int processes[], int
int bt, int at[], int Cid);
int wt[n]; float tat[n];
int total_wt = 0, total_tat = 0;
void AverageTimeJFPremptive(int processes[], int bt[], int at[], int
int total_wt, int total_tat);
printf("n JFP (preemptive scheduling)n");
printf("Average Arrival Time / Total Completion Time/n");
printf("Average Waiting Time/n");
for (int i=0; i<n; i++)
    total_wt += wt[i], total_tat += tat[i];
printf("Total waiting time = %d, processes[i].at[i], bt[i],
at[i], totalCid, wt[i]);\n");
float avg_wt = (float) total_wt / n;
float avg_tat = (float) total_tat / n;
printf("Avg_wt = %f/n, avg_wt = %f\n", avg_wt);
printf("Avg_tat = %f/n, avg_tat = %f\n", avg_tat);

int main() {
    int processes[10], burst_time[10], arrival_time[10];
}

```

	Page No. _____	VOOMA
--	----------------	-------

```

completion_time[n]; n;
printf("Enter no. of processes"); 
scanf("%d", &n);
printf("Enter arrival time and burst time for each process");
for (int i=0; i<n; i++) {
    printf("Arrival time of process-%d: ", i+1);
    scanf("%d", &arrival_time[i]);
    printf("Burst time of process-%d: ", i+1);
    scanf("%d", &burst_time[i]);
    processes[i] = i+1;
}

completion_time[0] = arrival_time[0] + burst_time[0];
for (int i=1; i<n; i++) {
    if (arrival_time[i] > completion_time[i-1]) {
        completion_time[i] = arrival_time[i] + burst_time[i];
    }
    else {
        completion_time[i] = completion_time[i-1] + burst_time[i];
    }
}

printf("FIFO Scheduling Process ID, Arrival Time, Burst Time,\nCompletion Time");
printf("Process ID, Arrival Time, Burst Time,\nCompletion Time");
printf("Process ID, Arrival Time, Completion Time");
return 0;
}

```

0.9849

Enter the number of processes: 3

Enter arrival time and burst time for each process

Arrival time: Arrival time of process 1 12.0

Burst time of process 1 2

Arrival time of process 2 1

Burst time of process 2 4

Arrival time of process 3 2

Burst time of process 3 5

PFC S

Process	Arrival Time	Burst Time	Completion Time	Turnaround Time	WT
1	0	2	2	2	0
2	1	4	6	5	1
3	2	5	11	9	7

Average waiting time = 1.6667

Average turnaround time = 4.6667

JIF (non-preemptive scheduling)

Process

Process	Arrival Time	Burst Time	Completion Time	TAT	WT
1	0	2	2	2	0
2	1	4	6	5	4
3	2	5	11	9	7

Average waiting time = 1.3333

Average turnaround time = 4.3333

## SJF (preemptive)

Process	Arrival Time	burst time	Completion Time	TAT	WT
1	0	2	2	2	0
2	1	4	9	8	4
3	2	3	5	3	0

$$\text{Average waiting time} = 1.333$$

$$\text{Average burst around time} = 4.333$$

Q.F.1  
8/8/24

LAB2: Write a C program to simulate the following CPU scheduling algorithm to find turnaround time and waiting time. → Priority → Round Robin (Experiment with different quantum sizes for RR algorithm)

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

void prioritySchedulingNonPreemptive(int n, int bt[], int priority[]) {
    int wt[n], tat[n], total_wt = 0, total_tat = 0;

    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - 1 - i; j++) {
            if (priority[j] > priority[j + 1]) {
                int temp = priority[j];
                priority[j] = priority[j + 1];
                priority[j + 1] = temp;

                temp = bt[j];
                bt[j] = bt[j + 1];
                bt[j + 1] = temp;
            }
        }
    }

    wt[0] = 0;

    for (int i = 1; i < n; i++) {
        wt[i] = wt[i - 1] + bt[i - 1];
        total_wt += wt[i];
    }

    for (int i = 0; i < n; i++) {
        tat[i] = bt[i] + wt[i];
        total_tat += tat[i];
    }

    printf("\nProcess\t Burst Time\t Priority\t Waiting Time\t Turnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\t%d\n", i + 1, bt[i], priority[i], wt[i], tat[i]);
    }
    printf("Average waiting time: %.2f\n", (float)total_wt / n);
    printf("Average turnaround time: %.2f\n", (float)total_tat / n);
}

void roundRobinScheduling(int n, int bt[], int quantum) {
    int remaining_bt[n], wt[n], tat[n], total_wt = 0, total_tat = 0;
```

INPUT:

```

    for (int i = 0; i < n; i++) {
        remaining_bt[i] = bt[i];
    }

    int t = 0;

    int completion_time[n];
    for (int i = 0; i < n; i++) {
        completion_time[i] = 0;
    }

    while (1) {
        int done = 1;

        for (int i = 0; i < n; i++) {
            if (remaining_bt[i] > 0) {
                done = 0;

                if (remaining_bt[i] > quantum) {
                    t += quantum;
                    remaining_bt[i] -= quantum;
                } else {
                    t += remaining_bt[i];
                    remaining_bt[i] = 0;
                }

                wt[i] = t - bt[i];

                tat[i] = t;
            }
        }

        if (done == 1)
            break;
    }

    printf("\nProcess\t Burst Time\t Waiting Time\t Turnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t %d\t %d\t %d\n", i + 1, bt[i], wt[i], tat[i]);
        total_wt += wt[i];
        total_tat += tat[i];
    }
}

```

```
    printf("Average turnaround time: %.2f\n", (float)total_tat / n);
}

int main() {
    int choice, n;

    do {
        printf("\nChoose the scheduling algorithm:\n");
        printf("1. Priority Scheduling (Non-preemptive)\n");
        printf("2. Round Robin Scheduling\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("\nEnter the number of processes for Priority Scheduling: ");
                scanf("%d", &n);

                int *bt1 = (int *)malloc(n * sizeof(int));
                int *priority1 = (int *)malloc(n * sizeof(int));

                printf("Enter burst time and priority for each process:\n");
                for (int i = 0; i < n; i++) {
                    printf("Process %d: ", i + 1);
                    scanf("%d %d", &bt1[i], &priority1[i]);
                }

                prioritySchedulingNonPreemptive(n, bt1, priority1);

                free(bt1);
                free(priority1);
                break;
            case 2:
                printf("\nEnter the number of processes for Round Robin Scheduling: ");
                scanf("%d", &n);

                int *bt2 = (int *)malloc(n * sizeof(int));
                int quantum;

                printf("Enter burst time for each process:\n");
                for (int i = 0; i < n; i++) {
                    printf("Process %d: ", i + 1);
                    scanf("%d", &bt2[i]);
                }

                printf("Enter time quantum for Round Robin Scheduling: ");
                scanf("%d", &quantum);
        }
    } while (choice != 3);
}
```

```

        for (int i = 0; i < n; i++) {
            printf("Process Id: ", i + 1);
            scanf("%d", &bt2[i]);
        }

        printf("Enter time quantum for Round Robin Scheduling: ");
        scanf("%d", &quantum);

        roundRobinScheduling(n, bt2, quantum);

        free(bt2);
        break;
    case 3:
        printf("\nExiting...\n");
        break;
    default:
        printf("\nInvalid choice. Please enter a valid option.\n");
    }
} while (choice != 3);

return 0;

```

**OUTPUT:**

```

Choose the scheduling algorithm:
1. Priority Scheduling
2. Round Robin Scheduling
3. Exit
Enter your choice: 1

Enter the number of processes for Priority Scheduling: 5
Enter burst time and priority for each process:
Process 1: 10
3
Process 2: 1
1
Process 3: 2
4
Process 4: 1
5
Process 5: 5
2

Process  Burst Time      Priority      Waiting Time      Turnaround Time
1          1                  1              0                1
2          5                  2              1                6
3          10                 3              6               16
4          2                  4              16              18
5          1                  5              18              19
Average waiting time: 8.20
Average turnaround time: 12.00

```

```

Choose the scheduling algorithm:
1. Priority Scheduling
2. Round Robin Scheduling
3. Exit
Enter your choice: 2

Enter the number of processes for Round Robin Scheduling: 3
Enter burst time for each process:
Process 1: 1
Process 2: 6
Process 3: 10
Enter time quantum for Round Robin Scheduling: 2

Process  Burst Time      Waiting Time      Turnaround Time
1          1                  0                1
2          6                  5                11
3          10                 7                17
Average waiting time: 4.00
Average turnaround time: 9.67

```

15/05/2022 Lab-2

With a C program to simulate an scheduling algorithm to find turn around time and waiting time.

→ Priority

→ Round Robin

INPUT

```
#include <stdio.h>
#include <stdlib.h>
void priorityScheduling(int n, int bt[], int priority[])
{
    int wt[n], tat[n], total_wt = 0, total_tat = 0;
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
            if (priority[j] > priority[j + 1])
            {
                int temp = priority[j];
                priority[j] = priority[j + 1];
                priority[j + 1] = temp;
                bt[j] = bt[j + 1];
                bt[j + 1] = temp;
            }
        }
    }
    wt[0] = 0;
    for (int i = 1; i < n; i++)
    {
        wt[i] = wt[i - 1] + bt[i - 1];
        total_wt += wt[i];
    }
    for (int i = 0; i < n; i++)
    {
        tat[i] = bt[i] + wt[i];
        total_tat += tat[i];
    }
}
```

```

printf("Product Burst Priority /t Waiting & turn around
Time\n");
for(int i=0; i<n; i++){
    printf("%d %d %d %d %d %d", i+1, bt[i], priority[i], wt[i],
    tat[i]);
}
printf("average waiting time = %f\n", (float) total_wt/n);
printf("average turn around time = %f\n", (float) total_tat/n);

```

```

void roundRobinScheduling (int n, int bt[], int quantum) {
    int remaining_bt[n], wt[n], tat[n], total_wt=0, total_tat=0;
    for (int i=0; i<n; i++) {
        remaining_bt[i] = bt[i];
    }
    int t=0;
    while (1) {
        int done=0;
        for (int i=0; i<n; i++) {
            if (remaining_bt[i]>0) {
                done=1;
                if (remaining_bt[i]>quantum) {
                    t+= quantum;
                    remaining_bt[i]-=quantum;
                    wt[i]= t - bt[i];
                } else {
                    t+= remaining_bt[i];
                    remaining_bt[i]=0;
                    wt[i]= t - bt[i];
                }
            }
        }
    }
}

```

```

tat[i] = t[i] - 2; // Adjustment for burst time
if (done == 1)
    break;
printf("Process %d Burst Time Waiting Time Turnaround Time\n");
for (int i = 0; i < n; i++) {
    printf("%d %d %d %d\n", i + 1, b[i], wt[i],
          tat[i]);
    total_wt += wt[i];
    total_tat = tat[i];
}
printf("Average waiting time = %.2f\n", (float) total_wt / n);
printf("Average turnaround time = %.2f\n", (float) total_tat / n);
}

int main() {
    int choice;
    do {
        printf("1. Round Robin Scheduling\n");
        printf("2. Priority Scheduling\n");
        printf("3. Exit\n");
        printf("Enter your choice");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter no. of processes for priority scheduling");
                scanf("%d", &n);
                if (n * pbtl = (int *) malloc (n * sizeof (int));
                if (n * priority = (int *) malloc (n * sizeof (int)));

```

M	T	W	F	S
Page No.		Date:	YOUVA	

```

printf("Enter burst time and priority for each process:\n");
for(int i=0; i<n; i++) {
    printf("Process ->d", i+1);
    scanf("%d", &btl[i]);
    &priority[i];
}
priorityDeccheduling(n, btl, priority);
free(btl);
free(priority);
break;

```

case 2:

```

printf("Enter the number of processes for round robin scheduling:");
scanf("%d", &n);
PWT * b2 = (int) malloc (n * sizeof(PWT));
int quantum;
for(i=0; i<n; i++) {
    printf("Enter burst time for each process:\n");
    for(j=0; j<n; j++) {
        printf("Process ->d", j+1);
        scanf("%d", &b2[i][j]);
    }
}

```

```

printf("Enter time quantum for Round Robin scheduling:");
scanf("%d", &quantum);
free(b2);
break;

```

case 3:

```

printf("exit");
break;

```

```

default:
printf("Invalid choice. Please enter a valid option\n");
3) while (choice != 3);
return 0;
}

```

OUTPUT:

choose the scheduling algorithm

1. Priority scheduling
2. Round Robin scheduling

Enter your choice:

Enter the number of processes for priority scheduling

Enter burst time and priority for each process

Process 1: 10 3

Process 2: 1 1

Process 3: 2 4

Process 4: 1 5

Process 5: 5 2

Process	Burst Time	Priority	Waiting Time	Turnaround Time
1	1	1	0	1
2	2	2	1	6
3	10	3	6	16
4	2	4	16	18
5	5	5	10	19

Average waiting time: 8.40

Average turnaround time: 12.00

Enter your choice: 2  
 Enter the number of processes for Round Robin Scheduling: 3  
 Enter burst time for each process:

User: 1

Process 1: 6

Process 2: 10

Process 3: 12

Enter time quantum for Round Robin Scheduling: 2

Process Burst Time Waiting Time Turnaround Time

1	1	0	13.0, 16.72, 16.72
---	---	---	--------------------

2	6	5	14.0, 16.72, 16.72
---	---	---	--------------------

3	10	7	17.0, 19.72, 19.72
---	----	---	--------------------

Average waiting time: 4.00

Average turnaround time: 9.00

Q.T.

LAB03:Write a C program to simulate multi-level queue scheduling algorithm considering the following scenario. All the processes in the system are divided into two categories– system processes and user processes. System processes are to be given higher priority than user processes. Use FCFS scheduling for the processes in each queue

INPUT:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #define MAX_PROCESSES 100
5
6 typedef struct {
7     int pid;
8     char type[10];
9     int arrival_time;
10    int burst_time;
11    int completion_time;
12    int turnaround_time;
13    int waiting_time;
14 } Process;
15
16 typedef struct {
17     Process processes[MAX_PROCESSES];
18     int front;
19     int rear;
20 } Queue;
21
22 void initQueue(Queue *q) {
23     q->front = -1;
24     q->rear = -1;
25 }
26
27 int isEmpty(Queue *q) {
28     return q->rear < q->front;
29 }
30
31 void enqueue(Queue *q, Process p) {
32     if (q->rear < MAX_PROCESSES - 1) {
33         q->processes[q->rear] = p;
34     } else {
35         printf("Queue is full!\n");
36     }
37 }
38
39 Process dequeue(Queue *q) {
40     if (!isEmpty(q)) {
41         return q->processes[q->front];
42     } else {
43         printf("Queue is empty!\n");
44         Process emptyProcess = {"P", "C", 0, 0, 0, 0, 0};
45         return emptyProcess;
46     }
47 }
48
49 void multilevelRoundRobinScheduling(Queue *systemQueue, Queue *userQueue,
50 int *currenttime, int *totalCompletionTime, int *totalTurnaroundTime, int *totalWaitingTime, int *totalProcesses) {
51     (*totalTypeArrival)(burst)(completion)(turnaround)(waiting)(0);
52     while (!isEmpty(systemQueue)) {
53         Process p = dequeue(systemQueue);
54         if (*currenttime <= p.arrival_time) {
55             *currenttime = p.arrival_time;
56             p.completion_time = *currenttime + p.burst_time;
57             p.turnaround_time = p.completion_time - p.arrival_time;
58             p.waiting_time = p.turnaround_time - p.burst_time;
59             *currenttime = p.completion_time;
60             *totalCompletionTime += p.completion_time;
61             *totalTurnaroundTime += p.turnaround_time;
62             *totalWaitingTime += p.waiting_time;
63             (*totalProcesses)++;
64             printf("Process %d completed at time %d\n", p.pid, p.completion_time);
65         }
66     }
67 }

```

```

17     p.pid, p.type, p.arrival_time, p.turnaround_time, p.completion_time, p.waiting_time);
18 }
19 while(!empty(userQueue)) {
20     Process p = dequeue(userQueue);
21     if (currenttime <= arrivalTime) {
22         currenttime = arrivalTime;
23     }
24     p.completion_time = currenttime + p.exec_time;
25     p.turnaround_time = p.completion_time - p.arrival_time;
26     p.waiting_time = p.turnaround_time - p.arrival_time;
27     totalCompletionTime += p.completion_time;
28     totalTurnaroundTime += p.turnaround_time;
29     totalWaitingTime += p.waiting_time;
30     totalProcesses++;
31     cout<<"Process ID: " << p.pid << endl;
32     cout<<p.pid<<, p.type<<, p.arrival_time<<, p.turnaround_time<<,
33     p.waiting_time<<endl;
34 }
35 }
36 int main() {
37     Queue systemQueue;
38     Queue userQueue;
39     InitQueue(systemQueue);
40     InitQueue(userQueue);
41     int numProcesses;
42     cout<<"Enter the number of processes: ";
43     cin>>numProcesses;
44     if (numProcesses > MAX PROCESSES) {
45         cout<<"Number of processes exceed the maximum limit of 1000." << endl;
46         return 1;
47     }
48     for (int i = 0; i < numProcesses; i++) {
49         Process p;
50         p.pid = i + 1;
51         cout<<"Enter type of process by [system/user]: "<< endl;
52         cout<<p.pid<<, p.type<<endl;
53         cout<<"Enter arrival time of process No: "<< endl;
54         cout<<p.pid<<, p.arrival_time<<endl;
55         cout<<"Enter burst time of process No: "<< endl;
56         cout<<p.pid<<, p.burst_time<<endl;
57         if ((p.type == "user") || (p.type == "user")) {
58             userQueue.enqueue(p);
59         } else {
60             systemQueue.enqueue(p);
61         }
62     }
63     currenttime = 0;
64     totalCompletionTime = 0;
65     totalTurnaroundTime = 0;
66     totalWaitingTime = 0;
67     totalProcesses = 0;
68     multivariantScheduling(systemQueue, userQueue, currenttime, totalCompletionTime, totalTurnaroundTime,
69     totalWaitingTime, totalProcesses);
70     if (totalProcesses > 0) {
71         cout<<"Average Completion Time: "<< float(totalCompletionTime / totalProcesses);
72         cout<<"Average Turnaround Time: "<< float(totalTurnaroundTime / totalProcesses);
73         cout<<"Average Waiting Time: "<< float(totalWaitingTime / totalProcesses);
74     }
75 }
76 }
```

OUTPUT:

```
Enter the number of processes: 5
Enter type of process 1 (system/user): system
Enter arrival time of process 1: 0
Enter burst time of process 1: 1
Enter type of process 2 (system/user): system
Enter arrival time of process 2: 1
Enter burst time of process 2: 2
Enter type of process 3 (system/user): user
Enter arrival time of process 3: 0
Enter burst time of process 3: 1
Enter type of process 4 (system/user): user
Enter arrival time of process 4: 1
Enter burst time of process 4: 2
Enter type of process 5 (system/user): system
Enter arrival time of process 5: 3
Enter burst time of process 5: 4
      PID    Type   Arrival   Burst   Completion   Turnaround   Waiting
1    system     0       1       1           1            0
2    system     1       2       3           2            0
5    system     3       4       7           4            0
3    user      0       1       8           8            7
4    user      1       2      10          9            7

Average Completion Time: 5.80
Average Turnaround Time: 4.80
Average Waiting Time: 2.80
```

3/6/20

LAB - 3

Write a C program to simulate multi-level queue scheduling algorithm considering the following scenario. All processes in the system are divided into two categories:

System processes and user processes. System processes are given higher priority than user processes.

use FCFS scheduling for the processes in each queue

INPUT : Number of processes in each queue

Includes stdio.h

Includes stdlib.h

Includes string.h

#define MAX\_PROCESSES 100

typedef struct

{int pid; arrival\_time, burst\_time, completion\_time,  
turnaround\_time, waiting\_time;

user\_type; } process

typedef struct

process processes [MAX\_processes];

int front, rear; } Queue;

void initQueue(Queue \*q){

q->front = 0; q->rear = -1; }

int isEmpty(Queue \*q){

return q->rear < q->front; }

void enqueue(Queue \*q, process p){

if (q->rear < MAX\_PROCESSES - 1){

q->processes[q->rear] = p;

else printf("Queue is full"); }

```

Process dequeue (Queue *q) {
    if (!IsEmpty(q)) {
        return q->processes[q->front + 1];
    } else {
        printf("Queue is Empty\n");
        Process emptyProcess = {0, 0, 0, 0, 0, 0};
        return emptyProcess;
    }
}

void multilevelQueueScheduling (Queue *systemQueue, Queue *userQueue, int *currentTime, int *totalCompletionTime,
                                int *totalTurnaroundTime, int *totalWaitingTime, int *total
                                burstTime) {
    while (!IsEmpty(systemQueue)) {
        Process p = dequeue (systemQueue);
        if (*currentTime <= p.arrivalTime) {
            *currentTime = p.arrivalTime;
            p.completionTime = *currentTime + p.burstTime;
            p.turnaroundTime = p.completionTime - p.arrivalTime;
            p.waitingTime = p.turnaroundTime - p.burstTime;
            *totalCompletionTime += p.completionTime;
            *totalTurnaroundTime += p.turnaroundTime;
            *totalWaitingTime += p.waitingTime;
            (*totalProcesses)++;
        }
    }
    while (!IsEmpty(userQueue)) {
        Process p = dequeue (userQueue);
        if (*currentTime <= p.arrivalTime) {
            *currentTime = p.arrivalTime;
            p.completionTime = *currentTime + p.burstTime;
        }
    }
}

```

$p_{i, j}$  turnaround time =  $p_i$ . completion time -  $p_i$ . arrival time  
 $p_{i, j}$  waiting time =  $p_i$ . turnaround time -  $p_i$ . burst time  
 \* total turnaround time =  $\sum p_i$ . completion time  
 \* total turnaround time =  $\sum p_i$ . turnaround time  
 \* total waiting time =  $\sum p_i$ . waiting time  
 \* total processes = 27

Put max no. of processes in system queue

One queue (system queue, user queue)

Init queue (system queue);  $\rightarrow$  allocation table & tail

put queue & user queue;

int n;

`printf("Enter no. of processes");`

`scanf("%d", &n);`

`if (n > MAX - 1) {`

`printf("Limit exceeded");`

`for (int i = 0; i < n; i++) {`

`processes[i] = 0;`

`p[i].id = i + 1;`

`printf("Enter type of process id, p.burst);`

`scanf("%d", &p[i].burst);`

`printf("Enter arrival time");`

`scanf("%d", &p[i].arrival_time);`

`printf("Enter burst time");`

`scanf("%d", &p[i].burst_time);`

`if (strcmp(p[i].type, "Burst") == 0) {`

`engrve(&system_queue, p[i]);`

```
if(strcmp(p->name, "user") == 0) {
```

    enqueue(&userQueue, p); break; // end if

```
    printf("Invalid process"); // end if
```

```
int waitingTime = 0, totalCompletionTime = 0, totalTurnaroundTime,
```

totalWaitingTime = 0, totalProcess = 0;

multilevelRoundRobin( &systemQueue, &userQueue, turnaroundTime,

, &totalCompletionTime, &totalTurnaroundTime, &totalWaitingTime,

, &totalProcesses);

```
} // end if (process == "user")
```

```
p->print("Average CT = " + 2 / m, avgCT = 2 / m average
```

WT = 2 / m); // print totalCompletionTime, totalProcesses

(float)totalTurnaroundTime / totalProcesses, (float)total

WaitingTime / totalProcesses);

```
return 0;
```

{ // end main() function for user input and output

    // end main() function for user input and output

    // end main() function for user input and output

    // end main() function for user input and output

    // end main() function for user input and output

    // end main() function for user input and output

    // end main() function for user input and output

    // end main() function for user input and output

    // end main() function for user input and output

    // end main() function for user input and output

    // end main() function for user input and output

    // end main() function for user input and output

OUTPUT

Enter the number of processes: 5

Enter type of process 1 (System/User): System

Enter arrival time of process 1: 0

Enter burst time of process 1: 1

Enter type of process 2 (System/User): System

Enter arrival time of process 2: 1

Enter burst time of process 2: 2

Enter type of process 3 (System/User): User

Enter arrival time of process 3: 0

Enter burst time of process 3: 3

Enter type of process 4 (System/User): User

Enter arrival time of process 4: 1

Enter burst time of process 4: 2

Enter type of process 5 (System/User): System

Enter arrival time of process 5: 3

Enter burst time of process 5: 4

PID	Type	Arrival	Burst	Completion	Turnaround	Waiting
1	System	0	1	1	1	0
2	System	1	2	3	2	0
5	System	3	4	7	4	0
3	User	0	1	8	8	7
4	User	1	2	10	9	7

Average completion time: 5.80 ms

Average turnaround time: 4.80 ms

Average waiting time = 2.80

## LAB04:

Write a C program to simulate Real-Time CPU Scheduling algorithms:

- a) Rate- Monotonic
- b) Earliest-deadline First
- c) Proportional scheduling

Input:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define MAX_TASKS 10
5
6 typedef struct {
7     int id;
8     int period;
9     int deadline;
10    int computation_time;
11    int remaining_time;
12    int completion_time;
13 } Task;
14
15 void rate_monotonic(Task tasks[], int num_tasks);
16 void earliest_deadline_first(Task tasks[], int num_tasks);
17 void proportional_scheduling(Task tasks[], int num_tasks);
18
19 int main() {
20     int num_tasks;
21     Task tasks[MAX_TASKS];
22
23     printf("Enter the number of tasks: ");
24     scanf("%d", &num_tasks);
25
26     printf("Enter the details of each task (ID, period, deadline, computation time):\n");
27     for (int i = 0; i < num_tasks; i++) {
28         printf("Task %d:\n", i + 1);
29         scanf("%d %d %d %d", &tasks[i].id, &tasks[i].period, &tasks[i].deadline, &tasks[i].computation_time);
30         tasks[i].remaining_time = tasks[i].computation_time;
31         tasks[i].completion_time = 0; // Initialize completion time
32     }
33
34     printf("Enter Inputs:\n");
35     for (int i = 0; i < num_tasks; i++) {
36         printf("Task %d (ID, Period, Deadline, Computation Time): ", i + 1, tasks[i].id, tasks[i].period, tasks[i].deadline, tasks[i].computation_time);
37     }
38 }
```

```

15    rate_monotonic(tasks, num_tasks);
16    earliest_deadline_first(tasks, num_tasks);
17    proportional_scheduling(tasks, num_tasks);
18
19    cout<<"/Completion times for the first two tasks:\n";
20    if (num_tasks == 2) {
21        cout<<"Task 1 (ID=0) Completion Time: " << tasks[0].id << ", tasks[0].completion_time);  

22    }
23    if (num_tasks == 3) {
24        cout<<"Task 2 (ID=0) Completion Time: " << tasks[1].id << ", tasks[1].completion_time);  

25    }
26
27    return 0;
28}
29
30void rate_monotonic(Task tasks[], int num_tasks) {
31    cout<<"\nRate-Monotonic Scheduling:\n";
32
33    for (int i = 0; i < num_tasks - 1; i++) {
34        for (int j = i + 1; j < num_tasks - 1; j++) {
35            if (tasks[i].period > tasks[j + 1].period) {
36                Task temp = tasks[i];
37                tasks[i] = tasks[j + 1];
38                tasks[j + 1] = temp;
39            }
40        }
41
42        int current_time = 0;
43        for (int l = 0; l < num_tasks; l++) {
44            tasks[l].completion_time = current_time + tasks[l].computation_time;
45            current_time = tasks[l].completion_time;
46            cout<<"Task " << l << " scheduled (Completion Time: " << tasks[l].id << ", tasks[l].completion_time);  

47        }
48    }
49}
50
51void earliest_deadline_first(Task tasks[], int num_tasks) {
52    cout<<"\nEarliest-Demand First Scheduling:\n";
53
54    for (int k = 0; k < num_tasks - 1; k++) {
55        for (int l = k + 1; l < num_tasks - 1; l++) {
56            if (tasks[k].deadline > tasks[l + 1].deadline) {
57                Task temp = tasks[k];
58                tasks[k] = tasks[l + 1];
59                tasks[l + 1] = temp;
60            }
61        }
62
63        int current_time = 0;
64        for (int i = 0; i < num_tasks; i++) {
65            tasks[i].completion_time = current_time + tasks[i].computation_time;
66            current_time = tasks[i].completion_time;
67            cout<<"Task " << i << " scheduled (Completion Time: " << tasks[i].id << ", tasks[i].completion_time);  

68        }
69    }
70}
71
72void proportional_scheduling(Task tasks[], int num_tasks) {
73    cout<<"\nProportional Scheduling:\n";
74
75    float total_period_inverse = 0;
76    for (int i = 0; i < num_tasks; i++) {
77        total_period_inverse += 1.0 / tasks[i].period;
78    }
79
80    for (int l = 0; l < num_tasks; l++) {
81        float proportion = (1.0 / tasks[l].period) / total_period_inverse;
82        cout<<"Task " << l << " gets " << proportion * 100 << "% of CPU time";  

83    }
84}

```

**Output:**

```
Enter the number of tasks: 3
Enter the details of each task (id, period, deadline, computation time):
Task 1: 1
5
5
2
Task 2: 2
10
10
1
Task 3: 3
7
7
3

User Inputs:
Task 1: ID=1, Period=5, Deadline=5, Computation Time=2
Task 2: ID=2, Period=10, Deadline=10, Computation Time=1
Task 3: ID=3, Period=7, Deadline=7, Computation Time=3

Rate-Monotonic Scheduling:
Task 1 scheduled (Completion Time: 2)
Task 3 scheduled (Completion Time: 5)
Task 2 scheduled (Completion Time: 6)

Earliest-Deadline First Scheduling:
Task 1 scheduled (Completion Time: 2)
Task 3 scheduled (Completion Time: 5)
Task 2 scheduled (Completion Time: 6)

Proportional Scheduling:
Task 1 gets 45.16% of CPU time
Task 3 gets 32.26% of CPU time
Task 2 gets 22.58% of CPU time
```

08/10/2024 LAB-4

Page No.:  
Date: YOUVRA

Write a C program to simulate Real Pm CPU alg scheduling  
algorithms.

a) Rate-monotonic task scheduling

b) Earliest deadline first

c) Proportional scheduling

INPUT:

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#define MAX_TASKS 10
```

```
typedef struct
```

```
{ int Id, period, deadline, weight, busy } Task;
```

```
int period;
```

```
int deadline;
```

```
int computation_time;
```

```
int remaining_time;
```

```
int completion_time; } Task;
```

```
void rate_monotonic(Task tasks[], int num_tasks);
```

```
void earliest_deadline_first(Task tasks[], int num_tasks);
```

```
void proportional_scheduling(Task tasks[], int num_tasks);
```

```
int main()
```

```
{ int num_tasks;
```

```
Task tasks[MAX_TASKS];
```

```
printf("Enter the number of tasks: ");
```

```
scanf("%d", &num_tasks);
```

```
printf("Enter details of each task (Id, period, deadline,
```

```
computation time): \n");
```

```

for (int i=0; i<num_tasks; i++) {
    printf("Task %d", i+1);
    scanf("%d %d %d %d", &task[i].id, &task[i].period,
    &task[i].deadline, &task[i].computation_time);
    task[i].remaining_time = task[i].computation_time;
    task[i].completion_time = 0;
}
printf("Enter Input:\n");
for (int i=0; i<num_tasks; i++) {
    printf("Task-%d. ID=%d Period=%d Deadline=%d Computation Time=%d\n", i+1, task[i].id, task[i].period, task[i].computation_time);
}
rate_monotonic(tasks, num_tasks);
earliest_deadline(first_C_tasks, num_tasks);
proportional_scheduling(tasks, num_tasks);
printf("In completion times for MC (first two tasks: n"), if (num_tasks >= 2) {
    printf("Task 1(n) = %d) completion Time=%d\n", task[0].id, task[0].computation_time);
    printf("Task 2(n) = %d) completion Time=%d\n", task[1].id, task[1].computation_time);
}
return 0;

```

```

void rate_monotonic(Task tasks[], int num_tasks) {
    printf("Rate Monotonic scheduling:\n");
    for (int i=0; i<num_tasks-1; i++) {
        for (int j=0; j<num_tasks-1; j++) {
            if (task[i].period < task[j].period)
                swap(task[i], task[j]);
        }
    }
}

```

if (task[i].period > tasks[j+1].period) {

Task temp = tasks[j];

tasks[j] = tasks[j+1];

tasks[j+1] = temp; }

int current\_time = 0;

for (int i = 0; i < num\_tasks; i++) {

tasks[task[i].completion\_time] = max(current\_time + task[i].computation\_time, current\_time);

current\_time = task[i].completion\_time;

printf("Task %d scheduled completion time = %d\n", task[i].id, task[i].completion\_time);

void earliestDeadline - first(Task tasks[], int num\_tasks)

void earliestDeadline First Scheduling: 1st,

for (int i = 0; i < num\_tasks - 1; i++) {

for (int j = 0; j < num\_tasks - i - 1; j++) {

if (tasks[j].deadline > tasks[j+1].deadline) {

Task temp = tasks[j];

tasks[j] = tasks[j+1];

tasks[j+1] = temp; }

int current\_time = 0;

for (int i = 0; i < num\_tasks; i++) {

tasks[task[i].completion\_time] = current\_time + task[i].computation\_time;

current\_time = task[i].completion\_time;

printf("Task %d scheduled completion time = %d\n", task[i].id, task[i].completion\_time);

void firstFit - first Fit Scheduling: 1st, 2nd,

```

void proportional_scheduling(Task tasks[], int num_tasks)
{
    printf("In Proportional Scheduling:\n");
    float total_period = 0;
    for (int i=0; i<num_tasks; i++)
    {
        total_period += 1.0 / tasks[i].period;
    }
    for (int i=0; i<num_tasks; i++)
    {
        tasks[i].period *= total_period;
    }
    float proportion = 0.0;
    for (int i=0; i<num_tasks; i++)
    {
        proportion *= (1.0 / tasks[i].period) / total_period;
        proportion += 1.0;
    }
    printf("Task %d gets a period of (%f seconds, task %d)\n",
        proportion * 1000);
}

```

OUTPUT

Enter the number of tasks: 3

Enter the details of each task (it's period, deadline, completion time):

Task 1:

5

5

2

Task 2:

10

10

1

Task 3:

7

User Inputs:

Task 1: ID = 1, Period = 5, Deadline = 5, Computation Time = 2

Task 2: ID = 2, Period = 10, Deadline = 10, Computation Time = 1

Task 3: ID = 3, Period = 7, Deadline = 7, Computation Time = 3

Rate - Monotonic Scheduling

Task 1 scheduled (Completion time = 2)

Task 2 scheduled (Completion time = 5)

Task 3 scheduled (Completion time = 6)

Earliest - Deadline First Scheduling

Task 1 scheduled (Completion time = 1)

Task 2 scheduled (Completion time = 5)

Task 3 scheduled (Completion time = 6)

Proportional Scheduling:

Task 1 gets  $15 \cdot 16 + \frac{1}{2}$  of CPU time

Task 2 gets  $32 \cdot 26 + \frac{1}{2}$  of CPU time

Task 3 gets  $22 \cdot 58 + \frac{1}{2}$  of CPU time

Sg.  
5/6/24

OSLAB05:Write a C program to simulate producer-consumer problem using semaphores.

**INPUT:**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <semaphore.h>
5
6 #define BUF_SIZE 10
7 #define MAX_ITHS 20
8
9 int buf[BUF_SIZE];
10 int cnt = 0;
11 int in = 0;
12 int out = 0;
13 int prod_cnt = 0;
14 int cons_cnt = 0;
15
16 pthread_mutex_t mtx;
17 pthread_cond_t cond_prod;
18 pthread_cond_t cond_cons;
19
20 void display_buffer() {
21     printf("Buffer: ");
22     for (int i = 0; i < out; ++i) {
23         printf("%d ", buf[(out - i) % BUF_SIZE]);
24     }
25     printf("\n");
26 }
27
28 void* prod(void* param) {
29     int prod_amount = *(int*)param;
30     int items_to_produce = prod_amount;
31
32     for (int i = 0; i < items_to_produce; ++i) {
33         int item = rand() % 100;
34
35         pthread_mutex_lock(&mtx);
36
37         while (out == BUF_SIZE) {
38             pthread_cond_wait(&cond_prod, &mtx);
39         }
40
41         buf[in] = item;
42         in = (in + 1) % BUF_SIZE;
43         cnt++;
44         prod_cnt++;
45         printf("Produced: %d\n", item);
46
47         display_buffer();
48
49         pthread_cond_signal(&cond_cons);
50 }
```

```

    pthread_cond_signal(&cond_poss);
    pthread_mutex_unlock(&mut);
    sleep(1);
}
return NULL;
}

void* receiver(void* param)
{
    while (1) {
        pthread_mutex_lock(&mut);
        if (recv_cue >= 3000000000) {
            pthread_mutex_unlock(&mut);
            break;
        }
        while (cue == 0) {
            pthread_cond_wait(&cond_cue, &mut);
        }
        display_buffer();
        int item = buf[cue];
        cue = (cue + 1) % BUF_SIZE;
        cue++;
        cue--;
        cout << cue << endl;
        cout << "CUE" << cue << endl;
        cout << "RECEIVED: " << item << endl;
        display_buffer();
        pthread_cond_signal(&cond_poss);
        pthread_mutex_unlock(&mut);
        sleep(1);
    }
    return NULL;
}

int main()
{
    pthread_t lid_posd, lid_cue;
    char choice;
    pthread_create(&lid_posd, NULL, posd, &choice);
    pthread_create(&lid_cue, NULL, cue, &choice);
    pthread_join(lid_posd, NULL);
    pthread_join(lid_cue, NULL);
    do {
        cout << "1)Send 2)Receive 3)Exit" << endl;
        cin << choice;
        if (choice == '1') {
            cout << "Enter file name: " << endl;
            string file_name;
            cin << file_name;
            cout << "File received successfully" << endl;
            cout << "Press 'q' to exit" << endl;
            cout << "Press 'r' to receive" << endl;
            cout << "Press 's' to send" << endl;
            cout << "Press 'e' to exit" << endl;
            cout << "Press 't' to receive from a connection component" << endl;
            cout << endl;
        }
        if (choice == '2') {
            cout << "Enter file name: " << endl;
            string file_name;
            cin << file_name;
            cout << "File received successfully" << endl;
            cout << "Press 'q' to exit" << endl;
            cout << endl;
        }
        if (choice == '3') {
            cout << "Enter file name: " << endl;
            string file_name;
            cin << file_name;
            cout << "File sent successfully" << endl;
            cout << endl;
        }
        if (choice == 'q') {
            cout << "Program exiting..." << endl;
            break;
        }
        if (choice == 'r') {
            cout << "File received successfully" << endl;
            cout << endl;
        }
        if (choice == 's') {
            cout << "File sent successfully" << endl;
            cout << endl;
        }
        if (choice == 'e') {
            cout << "Program exiting..." << endl;
            break;
        }
        if (choice == 't') {
            cout << "File received successfully" << endl;
            cout << endl;
        }
    } while (choice != 'q');
    pthread_exit(NULL);
}

```

## OUTPUT:

```
Menu:  
1. Start Production and Consumption  
2. Exit  
Enter your choice: 1  
Enter the number of items to produce: 5  
Produced: 41  
Buffer: [41 ]  
Buffer: [41 ]  
Consumed: 41  
Buffer: []  
Produced: 34  
Buffer: [34 ]  
Buffer: [34 ]  
Consumed: 34  
Buffer: []  
Produced: 69  
Buffer: [69 ]  
Produced: 78  
Buffer: [69 78 ]  
Produced: 62  
Buffer: [69 78 62 ]  
Buffer: [69 78 62 ]  
Consumed: 69  
Buffer: [78 62 ]  
Buffer: [78 62 ]  
Consumed: 78  
Buffer: [62 ]  
Buffer: [62 ]  
Consumed: 62  
Buffer: []
```

12/06/20

LAB-05

Write a C program to simulate producer-consumer problem  
using Semaphores

INPUTS:

#include <stdio.h>

#include <stdlib.h>

#include <semaphore.h>

#include <unistd.h>

#define BUF\_SIZE 10

#define MAX\_ITEMS 20

```
int buf[BUF_SIZE], cnt = 0, pn = 0, bn = 0, prod = 0,
```

cons = 0;

pmread = mutex\_t, mtx;

pmread = cond\_t, cond\_prod;

pmread = cond\_t, cond\_cons;

void display\_buffer()

printf("Buffer: [");

```
for (int i = 0; i < (cnt - 1); i++) {
```

printf("%d", buf[(cnt - 1) - i]);

}

printf("%d]", cnt);

void \*prod(void \*param)

int \*prod\_amount = (int \*) param;

int items\_to\_produce = \*prod\_amount;

```
for (int i = 0; i < items_to_produce; i++) {
```

int item = rand() + 100; // item will be between 100 & 150

pthread\_mutex\_lock(&mtx); // lock

while (cnt == BUF\_SIZE)

{ pthread\_cond\_wait(&cond, &mtx); }

buf[ptr] = item; // item will be between 100 & 150

ptr = (ptr + 1) % BUF\_SIZE

cnt++;

prod = cnt++; // now return to cond;

printf("Product: %d\n", item);

display\_buffer();

pthread\_cond\_signal(&cond); // now tag

pthread\_mutex\_unlock(&mtx);

sleep(rand() + 2); // wait real

{ // now it is the return boundary

item = NULL; // now done

}: // both can happen at the same time

void \* cond(void \* param){

while (true) // thread no. 1 will do this

pthread\_mutex\_lock(&mtx);

if (cnts - int >= MAX\_WAIT) {

{ pthread\_mutex\_unlock(&mtx);  
break; }

}

while (cnts == 0) {

pthread\_cond\_wait(&cond, &mtx); //

display\_buffer();

```

int i_km = buf[cout], v[100] {char v[100];};
out = cout + i_km - RBUF - R25; // return memory
int --; // CTRU read offset from RBUF
v[n] = v[i]++; // Read from memory
printf("consumed = %d\n", i_km);
leaving buffer(); // RBUF + CTRU = R25
    v[mread - vnd - signal < cond - prod];
    p[mread - mindex - unit] = LmIndex;
    sleep(rand() * .2); // wait 100ms
return Nuc; // completed job
}

int main() {
    // read from memory
    p[mread - vnd - prod] = vnd - cons;
    char choice; // Get character from user
    p[mread - mindex - int] = & mIndex, Nuc);
    p[mread - vnd - vnd + 4 * vnd - prod, Nuc];
    p[mread - vnd - int] = vnd - cons, Nuc];
    do {
        printf("Enter Nuc n. & start production and consumption\n");
        printf("Enter your choice: "); // read from user
        scanf("%c", &choice); // read from user
        switch (choice) {
            case '1': {
                p[vnd] = prod - amount; // read from user
                printf("Enter the number of items to produce: "); // read from user
            }
        }
    }
}

```

M	T	W	T	F	S
Page No.:					YOUVA
Date:					

scanf("%d", &prod\_amount);

p->read = create(L10d - prod, N12, prod, &prod\_amount);  
& prod\_amount);

p->read = create(L10d - cons, N12, cons, N12);

p->read = join(L10d - prod, N12);

p->read = join(L10d - cons, N12);

printf("Production & Consumption completed\n");  
case 2:

printf("Exiting...\n");  
break;

default:

printf("Invalid choice. Please try again.\n");

switch(choice) {

p->read = multx = destroy(L10d);

p->read = cons = destroy(L10d - prod);

p->read = cons = destroy(L10d - cons);

return 0;

DVTPVI

Menu

1. Start Production and Consumption

2. Exit

Enter your choice:

Enter the number of items to produce: 5

produced: 41

buffer: [41]

Buffer: [41]

consumed: 41

buffer: [ ]

produced: 34

buffer: [34]

buffer: [34]

consumed: 34

buffer: [ ]

produced: 69

buffer: [69]

buffer: [69]

consumed: 69

produced: 78

buffer: [69 + 78]

produced: 62

buffer: [69 + 78 + 62]

buffer: [69 + 78 + 62]

consumed: 69

buffer: [78 + 62]

buffer: [78 + 62]

consumed: 78

buffer: [62]

buffer: [62]

consumed: 62

buffer: [ ]

Sp. 1  
Sp. 1  
12/6/24

## LAB06:

Write a C program to simulate the concept of dining philosophers

INPUT:

```
1 #include <pthread.h>
2 #include <semaphore.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <unistd.h>
6 #include <time.h>
7 #define THINKING 2
8 #define HUNGRY 1
9 #define EATING 0
10 #define LEFT (phnum + N - 1) % N
11 #define RIGHT (phnum + 1) % N
12 int N;
13 int *state;
14 int *phil;
15 int running_time = 10;
16 time_t start_time;
17 sem_t mutex;
18 sem_t *S;
19 void test(int phnum) {
20     if (state[phnum] == HUNGRY
21         && state[LEFT] != EATING
22         && state[RIGHT] != EATING) {
23         state[phnum] = EATING;
24         sleep(2);
25         printf("Philosopher %d takes fork %d and %d\n",
26                phnum + 1, LEFT + 1, phnum + 1);
27         printf("Philosopher %d is Eating\n", phnum + 1);
28         sem_post(&S[phnum]);
29     }
30 }
31 void take_fork(int phnum) {
32     sem_wait(&mutex);
33     state[phnum] = HUNGRY;
34     printf("Philosopher %d is Hungry\n", phnum + 1);
35     test(phnum);
36     sem_post(&mutex);
37     sem_wait(&S[phnum]);
38     sleep(1);
```

```
40 void put_fork(int phnum) {
41     sem_wait(&mutex);
42     state[phnum] = THINKING;
43     printf("Philosopher %d putting fork %d and %d down\n",
44            phnum + 1, LEFT + 1, phnum + 1);
45     printf("Philosopher %d is thinking\n", phnum + 1);
46     test(LEFT);
47     test(RIGHT);
48     sem_post(&mutex);
49 }
50
51 void* philosopher(void* num) {
52     int* i = num;
53     while (1) {
54         if (difftime(time(NULL), start_time) >= running_time) {
55             break;
56         }
57         sleep(1);
58         take_fork(*i);
59         sleep(0);
60         put_fork(*i);
61     }
62     return NULL;
63 }
64
65 int main() {
66     printf("Enter the number of philosophers: ");
67     scanf("%d", &N);
68     state = (int *)malloc(N * sizeof(int));
69     phil = (int *)malloc(N * sizeof(int));
70     S = (sem_t *)malloc(N * sizeof(sem_t));
71     for (int i = 0; i < N; i++) {
72         phil[i] = i;
73     }
74     pthread_t thread_id[N];
75     sem_init(&mutex, 0, 1);
76     for (int i = 0; i < N; i++) {
```

```
56     break;
57 }
58 sleep(1);
59 take_fork(*i);
60 sleep(0);
61 put_fork(*i);
62 }
63 return NULL;
64 }
65
66 int main() {
67     printf("Enter the number of philosophers: ");
68     scanf("%d", &N);
69     state = (int *)malloc(N * sizeof(int));
70     phil = (int *)malloc(N * sizeof(int));
71     S = (sem_t *)malloc(N * sizeof(sem_t));
72     for (int i = 0; i < N; i++) {
73         phil[i] = i;
74     }
75     pthread_t thread_id[N];
76     sem_init(&mutex, 0, 1);
77     for (int i = 0; i < N; i++) {
78         sem_init(&S[i], 0, 0);
79     }
80     start_time = time(NULL);
81     for (int i = 0; i < N; i++) {
82         pthread_create(&thread_id[i], NULL, philosopher, &phil[i]);
83         printf("Philosopher %d is thinking\n", i + 1);
84     }
85     for (int i = 0; i < N; i++) {
86         pthread_join(thread_id[i], NULL);
87     }
88     free(state);
89     free(phil);
90     free(S);
91     return 0;
92 }
```

OUTPUT:

```
Enter the number of philosophers: 5
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 3 is Hungry
Philosopher 4 is Hungry
Philosopher 2 is Hungry
Philosopher 1 is Hungry
Philosopher 5 is Hungry
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinking
Philosopher 4 takes fork 3 and 4
Philosopher 4 is Eating
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
Philosopher 4 putting fork 3 and 4 down
Philosopher 4 is thinking
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 5 is Hungry
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is thinking
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eating
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinking
Philosopher 2 putting fork 1 and 2 down
Philosopher 2 is thinking
```

19/6/24 4AB-06

Page No. \_\_\_\_\_ Date: \_\_\_\_\_ YOUVA

Write a program to simulate the concept of Dining Philosophers problem

INPUT:

#include <phrread.h>

#include <semaphore.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <stropts.h>

#define THINKING 2

#define HUNGRY 1

#define EATING 0

#define LEFT (phnum + N - 1) % N

#define RIGHT (phnum + 1) % N

int N, \* state, \* phil, thinking, bone = 0;

time\_t start\_time;

sem\_t mutex, \* s;

void hot\_c(int phnum){

if (\*state[phnum] == HUNGRY & & state[LEFT] != EATING)

&& state[RIGHT] != EATING) state[phnum] = EATING;

sleep(2);

printf("Philosopher %d takes fork %d and %d in; phnum=%d, LEFT=%d, phnum=%d);\n",

printf("Philosopher %d is eating in; phnum=%d);\n",

sem\_post(s[phnum]); sleep(6); s[phnum] = 0;

void take\_fork(int phnum){

```

Date: 20-09-2018 YOUNG
sem, wait (Semaphore) is handle of message to share
state [phnum] = NUNGRY;
print("Philosopher " i " is hungry", phnum);
test & (phnum);
sem - post (sem mutex);
sem, wait (& s [phnum]);
sleep (1);
void put-fork (int phnum);
sem - wait (& mutex);
state [phnum] = THINKING;
print("Philosopher " i " is putting fork " i " and " j " down");
phnum++, LEFT + 1; phnum + 1);
print("Philosopher " i " is thinking", phnum);
test (LEFT); test (RIGHT); sem post (& mutex);
void philosopher(void * num);
int i = num;
while (1) {
    if (CLOCKTIMEOUT, start_time) running_time
        break; sleep (1); take-fork (i); sleep ();
        put-fork (i); sleep (1);
    no turn NULL;
}

int main() {
    int n;
    printf("Enter the number of philosophers: ");
    scanf("%d", &n);
    state = (int *) malloc (N * sizeof (int));
    Phil = (int *) malloc (N * sizeof (int));
}

```

```

s = (Sem_t*) malloc (N * sizeof (Sem_t));
for (int i = 0; i < N; i++) {
    Phil[i] = i;
    pMyread_t myread = i & CN; if (sem_init (&myread, 1);
    for (int i = 0; i < N; i++) {
        sem_init (&LS[i], 0, 0);
    }
    start_time = time (NULL);
    for (int i = 0; i < N; i++) {
        pMyread_create (&myread[i], NULL, philosopher, 2);
        printf ("Philosopher %d is thinking (%d)\n", i + 1, i);
        for (int j = 0; j < N; j++) {
            pMyread -> pin (thread[i], NULL);
        }
        free (phil);
        printf ("%d philosopher is hungry\n", i + 1);
        return 0;
    }
}

```

## OUTPUT:

Enter the number of philosophers: 5  
 Philosopher 1 is thinking  
 Philosopher 2 is thinking  
 Philosopher 3 is thinking  
 Philosopher 4 is thinking  
 Philosopher 5 is thinking  
 Philosopher 3 is hungry  
 Philosopher 4 is hungry  
 Philosopher 2 is hungry  
 Philosopher 1 is hungry

Philosopher 5 is hungry

Philosopher 5 is taking fork 5 and 2, 5

Philosopher 5 is eating

Philosopher 5 putting fork 5 and 2 down

Philosopher 5 is thinking

Philosopher 4 takes fork 3 and 4

Philosopher 4 is eating

Philosopher 4 takes fork 3 and 4

Philosopher 4 is eating

Philosopher 4 putting fork 3 and 4 down

Philosopher 4 is thinking

Philosopher 3 takes fork 2 and 3

Philosopher 3 is eating

Philosopher 3 is thinking

Philosopher 1 putting fork 3 and 1 down

Philosopher 1 is thinking

Philosopher 5 takes fork 4 and 5

Philosopher 5 is eating

Philosopher 3 is putting fork 2 and 3 down

Philosopher 3 is thinking

Philosopher 2 takes fork 1 and 2

Philosopher 2 is eating

Philosopher 5 putting fork 4 and 5 down

Philosopher 5 is thinking

Philosopher 2 putting fork 1 and 2 down

Philosopher 2 is thinking

## LAB07:

Write a C program to simulate Bankers algorithm for purpose of deadlock avoidance

INPUT:

```
1 #include <stdio.h>
2 #include <cs50.h>
3 void calculateNeed(int need[10][10], int max[10][10], int allot[10][10], int P, int R) {
4     for (int i = 0; i < P; i++) {
5         for (int j = 0; j < R; j++) {
6             need[i][j] = max[i][j] - allot[i][j];
7         }
8     }
9 }
10 bool isSafe(int processes[], int avail[], int max[10][10], int allot[10][10], int P, int R) {
11     int need[10][10];
12     calculateNeed(need, max, allot, P, R);
13     bool finish[10] = {0};
14     int safeSeq[10];
15     int work[10];
16     for (int i = 0; i < R; i++) {
17         work[i] = avail[i];
18     }
19     int count = 0;
20     while (count < P) {
21         bool found = false;
22         for (int p = 0; p < P; p++) {
23             if (finish[p] == 0) {
24                 int j;
25                 for (j = 0; j < R; j++) {
26                     if (need[p][j] > work[j])
27                         break;
28                 }
29                 if (j == R) {
30                     for (int k = 0; k < R; k++)
31                         work[k] += allot[p][k];
32                     safeSeq[count++] = p;
33                     finish[p] = 1;
34                     found = true;
35                 }
36             }
37         }
38         if (found == false) {
39             printf("System is not in safe state\n");
40             return false;
41         }
42     }
43     printf("SYSTEM IS IN SAFE STATE\n");
44     printf("The Safe Sequence is -- ");
45     for (int i = 0; i < P; i++) {
46         printf("%d ", safeSeq[i]);
47     }
48     printf("\n");
49     return true;
50 }
51 bool requestResource(int processes[], int avail[], int max[10][10], int allot[10][10], int P, int R, int pid, int request[]) {
```

```

92     processes[i] = i;
93
94     int avail[10];
95     int max[10][10];
96     int allot[10][10];
97
98     for (int i = 0; i < P; i++) {
99         printf("Enter allocation for P%d: ", i);
100        for (int j = 0; j < R; j++) {
101            scanf("%d", &allot[i][j]);
102        }
103        printf("Enter max for P%d: ", i);
104        for (int j = 0; j < R; j++) {
105            scanf("%d", &max[i][j]);
106        }
107    }
108    printf("Enter available resources: ");
109    for (int i = 0; i < R; i++) {
110        scanf("%d", &avail[i]);
111    }
112    int need[10][10];
113    calculateNeed(need, max, allot, P, R);
114    printf("\nProcess\tAllocation\tMax\t\tNeed\n");
115    for (int i = 0; i < P; i++) {
116        printf("P%d\t", i);
117        for (int j = 0; j < R; j++) {
118            printf("%d\t", allot[i][j]);
119        }
120        printf("\t\t");
121        for (int j = 0; j < R; j++) {
122            printf("%d\t", max[i][j]);
123        }
124        printf("\t\t");
125        for (int j = 0; j < R; j++) {
126            printf("%d\t", need[i][j]);
127        }
128        printf("\n");
129    }
130    isSafe(processes, avail, max, allot, P, R);
131    int pid, request[10];
132    printf("Enter PID for new request: ");
133    scanf("%d", &pid);
134    printf("Enter request for resources: ");
135    for (int i = 0; i < R; i++) {
136        scanf("%d", &request[i]);
137    }
138    requestResource(processes, avail, max, allot, P, R, pid, request);
139
140    return 0;
}

```

OUTPUT:

```

Enter number of processes: 5
Enter number of resources: 3
Enter allocation for P0: 0
1
0
Enter max for P0: 7
5
3
Enter allocation for P1: 2
0
0
Enter max for P1: 3
2
2
Enter allocation for P2: 3
0
2
Enter max for P2: 9
8
2
Enter allocation for P3: 2
1
1
Enter max for P3: 2
2
2
Enter allocation for P4: 0
0
2
Enter max for P4: 4
3
3
Enter available resources: 3
3
2

Process Allocation      Max          Need
P0    0 1 0      7 5 3      7 4 3
P1    2 0 0      3 2 2      1 2 2
P2    3 0 2      9 0 2      6 0 0
P3    2 1 1      2 2 2      0 1 1
P4    0 0 2      4 3 3      4 3 1

SYSTEM IS IN SAFE STATE
The Safe Sequence is -- P1 P3 P4 P0 P2
Enter PID for new request: 1
Enter request for resources: 1
0
2

SYSTEM IS IN SAFE STATE
The Safe Sequence is -- P1 P3 P4 P0 P2
Resources have been allocated successfully

```

LAB-07

YOUVA

Write a C program to simulate Banker's algorithm for the handling of deadlock avoidance.

#include <stdio.h>

#include <stdlib.h>

void calculateNeed (int need[5][10], int max[5][10], int allocation[5][10],  
int P, int R);

for (int p=0; p<P; p++) {

for (int j=0; j<R; j++) {

need[i][j] = max[i][j] - allocation[i][j];

bool isSafe (int processes, int available[], int max[5], int allocation[5][10], int P, int R);

int need[10][10];

int allocation[10][10], max[10][10], P, R;

int available[10] = {0};

int safetySeq[10];

for (int i=0; i<P; i++) {

work[i] = available[i];

int count = 0; while (count < P) {

if (work[i] >= need[i]) {

not found = false;

for (int p = 0; p < P; p++) {

if (work[p] >= need[p]) {

int j; if (j < R) {

for (int i=0; i<R; i++) {

if (need[p][i] > work[i]) {

```

    if (j == CR) {
        pos (int k = 0; k < R; k++)
            work [k] = allot [p1[k]];
        safeSeq [count++ - 1] = p;
        if (safeSeq [count - 1] == 1) {
            found = true;
        }
    }
    if (found == false) {
        printf ("System is not in safe state\n");
        return false;
    }
    printf ("No safe sequence in \"%s\".\n", j);
    for (int i = 0; i < count; i++)
        printf ("%d ", safeSeq[i]);
    printf ("\n");
    return true;
}

bool request (char *intProcess[], int avail[], int max[], int id,
    int allot[], int p1[], int R, int pid, int requests[])
{
    int need [10];
    calculateNeed (need, max, allot, p1, R);
    for (int i = 0; i < R; i++)
        if (request[i] > need[pid]) {
            printf ("Error: Process has exceeded its maximum limit\n");
            return false;
        }
    for (int i = 0; i < R; i++)
        if (request[i] > avail[i]) {
            printf ("Resources are not available, process must wait\n");
        }
}

```

```

return false;
for (int i=0; i<R; i++) {
    avail[i] = request[i];
    allot[i][i] = request[i];
    need[i][i] = request[i];
}
if (isSafeProcess, avail[i], max, allot, P, R) == false {
    avail[i] += request[i];
    allot[i][i] += request[i];
    need[i][i] += request[i];
    printf("System is not in safe state after allocation %d\n");
    return false;
}
printf("Resources have been allocated successfully\n");
return true;

```

int main()

```

int P, R;
printf("Enter number of processes and resources : ");
scanf("%d %d", &P, &R);
int process[10], avail[10][10], max[10][10], allot[10][10];
for (int i=0; i<P; i++) {
    process[i] = i+1;
    need[10][i] = P;
    request[i] = 0;
}
printf("Enter Allocation for P+R %d, %d\n");
for (int i=0; i<P; i++) {
    for (int j=0; j<R; j++) {
        scanf("%d", &allot[i][j]);
    }
}
printf("Enter max for P+R %d\n");
for (int j=0; j<R; j++) {
    scanf("%d", &max[10][j]);
}

```

```

printf("Enter available resources\n");
for (i=0; i<R; i++) {
    scanf("%d", &avail[i]);
}
calculateNeed (need, max, avail, P, R);
printf("Process Allocation to Maximal Need\n");
for (j=0; j<P; j++) {
    printf("P %d I %d, ", j, need[j]);
    for (i=0; i<R; i++) {
        printf("R %d, ", i);
    }
    printf("\n");
}
printf("Safe Sequence: ");
for (j=0; j<P; j++) {
    printf("%d ", need[j]);
}
printf("\n");
isSafe (processes, avail, max, allot, P, R);
printf("Process Requests: ");
printf("Enter P ID and Request for new request\n");
scanf("%d", &pid);
printf("Enter request for resource\n");
for (int i=0; i<K; i++) {
    scanf("%d", &request[i]);
}
requestResource (processes, avail, max, allot, P, R, pid, request);
return 0;
}

```

$$D \vee F P \vee F$$

Enter number of processes: 5  
Enter number of threads per process: 3

Enter number of resources = 3

Enter allocation for  $P_0 = 0$

6

Enter max for  $P_0$  : 7

5

3

Enter allocation for P1: 2

1

1

10

10

10

10

1

10

104

104

104

1

1

- 4 -

10

1  
Enter allocation for  $P_4$ : 0

0

2

Enter max for  $P_4$ : 4

3

3

Enter available resources: 3

3

2

Process Allocation	Max	Need
$P_0$ 0 1 0	2 2 3	2 4 3
$P_1$ 2 0 0	3 2 2	1 2 2
$P_2$ 3 0 2	4 0 2	6 0 0
$P_3$ 2 1 1	2 2 2	0 1 1
$P_4$ 0 0 2	4 3 3	8 4 3

SYSTEM IS IN SAFE STATE

The Safe Sequence is -  $P_1 P_3 P_4 P_0 P_2$

Enter PID for new request: 1

Enter request for resource: 1

2

SYSTEM IS IN SAFE STATE

The Safe Sequence is -  $P_1 P_3 P_4 P_0 P_2$

Resources have been allocated successfully

## LAB08:

Write a C program to simulate deadlock detection

INPUT:

```
1 #include <stdio.h>
2 #include <stdbool.h>
3
4 bool isLessThanOrEqual(int request[], int work[], int R) {
5     for (int i = 0; i < R; i++) {
6         if (request[i] > work[i]) {
7             return false;
8         }
9     }
10    return true;
11}
12
13 void addVectors(int work[], int allocation[], int R) {
14     for (int i = 0; i < R; i++) {
15         work[i] += allocation[i];
16     }
17}
18
19 void printState(bool finish[], int P) {
20     printf("Finish vector: ");
21     for (int i = 0; i < P; i++) {
22         printf("%s ", finish[i] ? "true" : "false");
23     }
24     printf("\n");
25}
26
27 void deadlockDetection(int allocation[][3], int request[][3], int available[], int P, int R) {
28     int work[R];
29     bool finish[P];
30     int sequence[P];
31     int index = 0;
32
33     for (int i = 0; i < R; i++) {
34         work[i] = available[i];
35     }
36     for (int i = 0; i < P; i++) {
37         bool nonzero = false;
38         for (int j = 0; j < R; j++) {
39             if (allocation[i][j] != 0) {
40                 nonzero = true;
41                 break;
42             }
43         }
44         finish[i] = !nonzero;
45     }
46
47     while (true) {
48         bool found = false;
49         for (int i = 0; i < P; i++) {
```

```

59         for (int i = 0; i < P; i++) {
60             if (!finish[i] && isLessThanOrEqual(request[i], work, R)) {
61                 addVectors(work, allocation[i], R);
62                 finish[i] = true;
63                 sequence[index++ = i];
64                 found = true;
65                 break;
66             }
67         }
68         if (!found) {
69             break;
70         }
71     }
72
73     bool deadlock = false;
74     for (int i = 0; i < P; i++) {
75         if (!finish[i]) {
76             printf("System is in deadlock, process P%d is deadlocked.\n", i);
77             deadlock = true;
78         }
79     }
80     if (!deadlock) {
81         printf("System is not in deadlock.\nSafe Sequence: ");
82         for (int i = 0; i < P; i++) {
83             printf("P%d ", sequence[i]);
84         }
85         printf("\n");
86     }
87 }
88
89 int main() {
90     int P, R;
91
92     printf("Enter the number of processes: ");
93     scanf("%d", &P);
94
95     printf("Enter the number of resource types: ");
96     scanf("%d", &R);
97
98     int allocation[P][R], request[P][R], available[R];
99
100    printf("Enter the Allocation Matrix:\n");
101    for (int i = 0; i < P; i++) {
102        printf("Process P%d:\n", i);
103        for (int j = 0; j < R; j++) {
104            printf("Resource %c: ", 'A' + j);
105            scanf("%d", &allocation[i][j]);
106        }
107    }
108
109    printf("Enter the Request Matrix:\n");
110    for (int i = 0; i < P; i++) {
111        printf("Process P%d:\n", i);
112        for (int j = 0; j < R; j++) {
113            printf("Resource %c: ", 'A' + j);
114            scanf("%d", &request[i][j]);
115        }
116    }
117
118    deadlockDetection(allocation, request, available, P, R);
119
120    return 0;
121}

```

OUTPUT:

```
Enter the number of processes: 5
Enter the number of resource types: 3
Enter the Allocation Matrix:
Process P0:
Resource A: 0
Resource B: 1
Resource C: 0
Process P1:
Resource A: 2
Resource B: 0
Resource C: 0
Process P2:
Resource A: 3
Resource B: 0
Resource C: 3
Process P3:
Resource A: 2
Resource B: 1
Resource C: 1
Process P4:
Resource A: 0
Resource B: 0
Resource C: 2
Enter the Request Matrix:
Process P0:
Resource A: 0
Resource B: 0
Resource C: 0
Process P1:
Resource A: 2
Resource B: 0
Resource C: 2
Process P2:
Resource A: 0
Resource B: 0
Resource C: 0
Process P3:
Resource A: 1
Resource B: 0
Resource C: 0
Process P4:
Resource A: 0
Resource B: 0
Resource C: 2
Enter the Available Resources:
Resource A: 0
Resource B: 0
Resource C: 0
System is not in deadlock.
Safe Sequence: P0 P2 P1 P3 P4
```

3/07/2024

Page No.

Date

YOUVA

Write a C program to simulate deadlock detection  
(NPUS)

#include <cs.h>

#include <stdlib.h>

bool isLessThanOrEqual(int request[3], int work[3], int R[3]) {  
 for (int i=0; i<R; i++) {

if (request[i] > work[i]) return false;

}  
 return true; }

void addRequest(int work[3], int allocation, int R[3]) {

for (int i=0; i<R; i++) {

work[i] += allocation[i];

void printState(char \*msg, int P) {

printf("%s", msg);

for (int i=0; i<P; i++) {

printf(" %d", work[i]);

printf("\n");

void deadlockDetection(int allocation[3][3], int request[3][3],

int work[3], sequence[3], index=0;

bool finish[3];

for (int i=0; i<R; i++) {

work[i] = availability[i];

for (int i=0; i<P; i++) {

bool nonzero=false;

for (int j=0; j<R; j++) {

```

if (allocation[i][j] == 0) {
    nonzero = true;
    break;
}

```

Date: \_\_\_\_\_  
Page: \_\_\_\_\_

```

if (nonzero == true; j++)
    while (true) { i++; if (i >= n - 1) break;
        bool found = false; if (i <= m - 1) break;
        for (int k = 0; k < p; k++) {
            if (!l.finish[i].isLessThanOrEqual(sequenc[i], m),
                R)) {

```

```

                addNotes(wrk, allocation[i], R);
                finish[i] = true; i = 0; break;
                sequence[i] = index + 1;
                found = true; i = 0; break;
            }
        }
    }

```

```

    if (found) {
        break;
    }
}
```

```

bool deadlock = false;
for (int i = 0; i < p; i++) {
    if (!l.finish[i]) {

```

~~printf("System is not in deadlock in soft scenario\nfor (int i = 0, p = p; i < p; i++) {\n printf("%d ", sequence[i]);\n printf("\n");\n}~~

```

int main() {
    int l, R;
    printf("Enter the number of processes: ");

```

```

scanf("%d", &P);
printf("Enter the number of resource types = ");
scanf("%d", &R);
int allocation[P][R];
printf("Enter the allocation matrix N x M = ");
for (int i=0; i<P; i++) {
    printf("Row %d: ", i+1);
    for (int j=0; j<R; j++) {
        printf("Resource %d: ", j+1);
        scanf("%d", &allocation[i][j]);
    }
}
printf("Enter the Request matrix = ");
for (int i=0; i<P; i++) {
    printf("Row %d: ", i+1);
    for (int j=0; j<R; j++) {
        printf("Request %d: ", j+1);
        scanf("%d", &request[i][j]);
    }
}
printf("Available Resources = ");
for (int i=0; i<R; i++) {
    printf("Resource %d: ", i+1);
    scanf("%d", &available[i]);
}
deadlockDetection(allocation, request, available, P, R);
return 0;
}

```

Deadlock detection  
Algorithm  
Implementation

Enter the number of processes: 5

Enter the number of resource types: 3

Enter the Allocation Matrix:

Process P0 P1 P2 P3 P4 Resource A 10 10 10 10 10

Resource B 0 0 0 0 0

Resource C 0 0 0 0 0

Process P1: 10 10 10 10 10

Resource A: 0 0 0 0 0

Resource B: 0 0 0 0 0

Resource C: 0 0 0 0 0

Process P2: 10 10 10 10 10

Resource A: 0 0 0 0 0

Resource B: 0 0 0 0 0

Resource C: 0 0 0 0 0

Process P3: 10 10 10 10 10

Resource A: 0 0 0 0 0

Resource B: 0 0 0 0 0

Resource C: 0 0 0 0 0

Process P4:

Resource A: 0

Resource B: 0

Resource C: 0

Enter the Request Matrix

Process P0:

Resource A: 0

Resource B: 0, not needed at moment of arrival

Resource C: 0, not needed at moment of arrival

Process P1:

Resource A: 2

Resource B: 0

Resource C: 2

Process P2:

Resource A: 0, enough available to start work

Resource B: 0

Resource C: 0

Process P3:

Resource A: 1

Resource B: 0

Resource C: 0, enough available to start work

Process P4:

Resource A: 0

Resource B: 0

Resource C: 2

Enter the Available Resources

Resource A: 0

Resource B: 0

Resource C: 0

begin is not in deadlock

safe sequence: P2 P1 P3 P4

LAB09: Write a C program to simulate the following contiguous memory techniques

- a)First Fit
- b)Best Fit
- c)Worst Fit

Input:

```
1 #include <stdio.h>
2 #include <limits.h>
3
4 void firstFit(int blockSize[], int m, int fileSize[], int n) {
5     int allocation[n];
6
7     for (int i = 0; i < n; i++) {
8         allocation[i] = -1;
9     }
10
11    for (int i = 0; i < n; i++) {
12        for (int j = 0; j < m; j++) {
13            if (blockSize[j] >= fileSize[i]) {
14                allocation[i] = j;
15                blockSize[j] -= fileSize[i];
16                break;
17            }
18        }
19    }
20
21    printf("Memory Management Scheme - First Fit\n");
22    printf("File_no\tfile_size\tBlock_no\tBlock_size\tfragment\n");
23    for (int i = 0; i < n; i++) {
24        printf("%d\t%d\t", i+1, fileSize[i]);
25        if (allocation[i] != -1) {
26            printf("%d\t%d\t", allocation[i]+1, blockSize[allocation[i]] );
27        } else {
28            printf("Not Allocated");
29        }
30    }
31    printf("\n");
32}
33
34 void bestFit(int blockSize[], int m, int fileSize[], int n) {
35     int allocation[n];
36
37     for (int i = 0; i < n; i++) {
38         allocation[i] = -1;
39     }
40
41     for (int i = 0; i < n; i++) {
42         int bestIdx = -1;
43         for (int j = 0; j < m; j++) {
44             if (blockSize[j] >= fileSize[i]) {
45                 if (bestIdx == -1 || blockSize[bestIdx] > blockSize[j]) {
46                     bestIdx = j;
47                 }
48             }
49         }
50     }
51 }
```

```

50         if (bestIdx != -1) {
51             allocation[i] = bestIdx;
52             blockSize[bestIdx] -= fileSize[i];
53         }
54     }
55 }
56
57 printf("Memory Management Scheme - Best Fit\n");
58 printf("%d\t%d\t%d\t%d\t%d\n", file_size, block_size);
59 for (int i = 0; i < n; i++) {
60     printf("%d\t%d\t%d\t%d\t%d", i+1, fileSize[i]);
61     if (allocation[i] != -1) {
62         printf("%d\t%d\t%d\t%d", allocation[i]+1, blockSize[allocation[i]], blockSize[allocation[i]]));
63     } else {
64         printf("Not Allocated");
65     }
66 }
67 printf("\n");
68 }
69
70 void worstFit(int blockSize[], int m, int fileSize[], int n) {
71     int allocation[n];
72
73     for (int i = 0; i < n; i++) {
74         allocation[i] = -1;
75     }
76
77     for (int i = 0; i < n; i++) {
78         int worstIdx = -1;
79         for (int j = 0; j < m; j++) {
80             if (blockSize[j] >= fileSize[i]) {
81                 if (worstIdx == -1 || blockSize[worstIdx] < blockSize[j]) {
82                     worstIdx = j;
83                 }
84             }
85         }
86
87         if (worstIdx != -1) {
88             allocation[i] = worstIdx;
89             blockSize[worstIdx] -= fileSize[i];
90         }
91     }
92
93     printf("Memory Management Scheme - Worst Fit\n");
94     printf("%d\t%d\t%d\t%d\t%d\n", file_size, block_size);
95     for (int i = 0; i < n; i++) {
96         printf("%d\t%d\t%d\t%d\t%d", i+1, fileSize[i]);
97         if (allocation[i] != -1) {
98             printf("%d\t%d\t%d\t%d", allocation[i]+1, blockSize[allocation[i]], blockSize[allocation[i]]));
99         }
100    }
101 }

```

```
98         printf("%d\t%d\t%d\n", allocation[i]+1, blockSize[allocation[i]], blockSize[sAllocation[i]]);
99     } else {
100         printf("Not Allocated\n");
101     }
102 }
103 printf("\n");
104 }
105
106 int main() {
107     int m, n;
108
109     printf("Enter the number of blocks: ");
110     scanf("%d", &m);
111     int blockSize[m];
112     printf("Enter the size of the blocks:\n");
113     for (int i = 0; i < m; i++) {
114         printf("Block %d: ", i+1);
115         scanf("%d", &blockSize[i]);
116     }
117
118     printf("Enter the number of files: ");
119     scanf("%d", &n);
120     int fileSize[n];
121     printf("Enter the size of the files:\n");
122     for (int i = 0; i < n; i++) {
123         printf("File %d: ", i+1);
124         scanf("%d", &fileSize[i]);
125     }
126
127     int blockSizeCopy[m];
128     for (int i = 0; i < m; i++) {
129         blockSizeCopy[i] = blockSize[i];
130     }
131
132     firstFit(blockSizeCopy, m, fileSize, n);
133
134     for (int i = 0; i < m; i++) {
135         blockSizeCopy[i] = blockSize[i];
136     }
137
138     bestFit(blockSizeCopy, m, fileSize, n);
139
140     for (int i = 0; i < m; i++) {
141         blockSizeCopy[i] = blockSize[i];
142     }
143
144     worstFit(blockSizeCopy, m, fileSize, n);
145     return 0;
146 }
```

Output:

```
Enter the number of blocks: 5
Enter the size of the blocks:
Block 1: 400
Block 2: 700
Block 3: 200
Block 4: 300
Block 5: 600
Enter the number of files: 4
Enter the size of the files:
File 1: 212
File 2: 517
File 3: 312
File 4: 526
Memory Management Scheme - First Fit
File_no File_size      Block_no      Block_size      Fragment
1       212            1             188            188
2       517            2             183            183
3       312            5             288            288
4       526            Not Allocated

Memory Management Scheme - Best Fit
File_no File_size      Block_no      Block_size      Fragment
1       212            4             88             88
2       517            5             83             83
3       312            1             88             88
4       526            2             174            174

Memory Management Scheme - Worst Fit
File_no File_size      Block_no      Block_size      Fragment
1       212            2             176            176
2       517            5             83             83
3       312            2             176            176
4       526            Not Allocated
```

03/07/20

Write a C program to simulate the following contiguous memory allocation techniques

- a) Worst Fit
- b) Best Fit
- c) First Fit

```
#include <stdio.h>
#include <limits.h>
void firstFit(int blockSize[], int m, int p[], int n) {
    printf("Allocation of blocks\n");
    for (int i = 0; i < n; i++) {
        int allocationIndex = -1;
        for (int j = 0; j < m; j++) {
            if (blockSize[j] >= p[i]) {
                allocationIndex = j;
                blockSize[j] -= p[i];
                break;
            }
        }
        printf("Block size %d is allocated at index %d\n", p[i], allocationIndex);
    }
    printf("\n");
}

printf("Memory Management Scheme - First Fit\n");
printf("File - void firstFit(int blockSize[], int m, int p[], int n)\n");
printf("Fragment\n");
for (int i = 0; i < n; i++) {
    printf(" %d ", allocationIndex[i]);
}
printf("\n");
blockSize[allocationIndex[i]] = blockSize[allocationIndex[i]];
else
    printf("Not allocated\n");
    printf("\n");
}
```

```

void bestfit (int blocksize[], int m, int psize[], int n) {
    int allocation[m];
    for (int i = 0; i < n; i++) {
        allocation[i] = -1;
    }
    for (int j = 0, p = n; j < m; j++) {
        if (bestIndex == -1) {
            if (blocksize[j] >= psize[i]) {
                if ((bestIndex == -1) || blocksize[bestIndex] > blocksize[j]) {
                    bestIndex = j;
                }
            }
        } else if (blocksize[bestIndex] >= psize[i]) {
            if ((bestIndex == -1) || blocksize[bestIndex] > blocksize[i]) {
                bestIndex = i;
            }
        }
    }
    allocation[bestIndex] = bestIndex;
    blocksize[bestIndex] -= psize[i];
    printf("Memory Management Scheme - best fit(%d);\n");
    printf("Allocated Block %d to Process %d\n");
    printf("Fragmentation:\n");
    for (int k = 0, i = 0; k < n; k++) {
        printf("%d + d1 + d2 + d3 + d4", k + 1, psize[i]);
        if (allocation[i] != -1) {
            printf(" + allocation[%d] = %d)\n", i, allocation[i]);
            printf("%d + d1 + d2 + d3 + d4 + d5 + d6 + d7 + d8 + d9 + d10, allocation[%d] +\n"
                   "blocksize[allocation[%d]], blocksize[allocation[%d]]);\n", i, allocation[i], i, allocation[i]);
        }
    }
    printf("Not allocated\n");
    printf("n");
}

```

```

void worstFit(int blockSize[], int m, int fileSize[], int n) {
    int allocation[m];
    for (int i=0; i<n; i++) {
        allocation[i] = -1;
    }
    for (int i=0; i<n; i++) {
        int worstIndex = -1;
        for (int j=0; j<m; j++) {
            if (blockSize[j] >= fileSize[i]) {
                if (allocation[j] == -1) {
                    allocation[i] = j;
                    worstIndex = j;
                    break;
                }
            }
        }
    }
}

```

```

if (worstIndex != -1) {
    allocation[worstIndex] = worstIndex;
    printf("Allocation of block %d to process %d\n", allocation[worstIndex], i+1);
    printf("block size %d assigned to process %d\n", blockSize[worstIndex], i+1);
}
printf("Memory Management Scheme - Worst Fit\n");
printf("File name\tBlock size\tBlock no\tBlock size\n");
printf("Fragmentation");

```

```

for (int i=0; i<n; i++) {
    printf("%d\t%d\t%d\t", i+1, fileSize[i], allocation[i]);
}

```

Worst Fit Allocation:

```

printf("Worst Fit Allocation:\n");
for (int i=0; i<n; i++) {
    printf("%d\t%d\t%d\t", i+1, allocation[i], blockSize[allocation[i]]);
}

```

```

printf("Not allocated\n");
printf("\n");
}

```

int main()

{ int m, n;

printf("Enter the number of blocks:");

scanf("%d", &m);

printf("Enter the size of the blocks:");

for (int i=0; i < m; i++)

printf("Block %d = %d\n",

i, scanf("%d", &BlockSize[i]));

printf("Enter the number of Plus:");

scanf("%d", &n);

int plus[0];

printf("Enter message of the Plus:");

for (int i=0; i < n; i++)

printf("Plus %d = %d\n", i, plus[i]);

scanf("%d", &plus[i]));

int blockSize[0];

(so (int i=0; i < m; i++) {

blockSize[i] = blockSize[i];

for (int i=0; i < m; i++) {

blockSize[i] = blockSize[i];

bestFit(blockSize, m, plus[0], n);

for (int i=0; i < m; i++) {

blockSize[i] = blockSize[i];

Worst Fit (block size by m, file size, m);

(m=100)

OUTPUT

Enter no. number of blocks: 5

Enter size of each blocks: 100

Block1: 100

Block2: 200

Block3: 200

Block4: 300

Block5: 600

Enter no. number of files: 4

Enter size of the file:

File1: 212

File2: 517

File3: 312

File4: 526

Memory Management Scheme - Worst Fit

File-no	File-Size	Block-no	block-size	Fragment
1	212	1	100	100
2	517	2	100	100
3	312	3	100	100
4	526	4	100	Not Allocated

Memory Management Scheme - First Fit			
F <sub>i</sub> l <sub>i</sub> -no	F <sub>i</sub> l <sub>i</sub> -size	Block-no	Fragment
1	212	1	188
2	517	2	183
3	312	5	288
4	526	Not allocated	

Memory Management Scheme - Best Fit			
F <sub>i</sub> l <sub>i</sub> -no	F <sub>i</sub> l <sub>i</sub> -size	Block-no	Fragment
1	212	4	88
2	517	5	83
3	312	1	87
4	526	2	114

Memory Management Scheme - Best Fit			
F <sub>i</sub> l <sub>i</sub> -no	F <sub>i</sub> l <sub>i</sub> -size	Block-no	Fragment
1	212	2	176
2	512	5	88
3	312	1	176
4	526	Not allocated	

S.P.J.  
3/2/24