Week - 5
(20 July 2023)
Experiment - 5

Question:

1) Write a C program to simulate Real-Time CPU Scheduling algorithms:
- **a)** Rate- Monotonic
- **b)** Earliest-deadline First
- **c)** Proportional scheduling

2) Write a C program to simulate producer-consumer problem using semaphores.

3) Write a C program to simulate the concept of Dining-Philosophers problem.

Program:

1) CPU Scheduling:

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

#define MAX_PROCESS 10

typedef struct {
   int id;
   int burst_time;
   float priority;
} Task;

int num_of_process;
int execution_time[MAX_PROCESS], period[MAX_PROCESS], remain_time[MAX_PROCESS],
deadline[MAX_PROCESS], remain_deadline[MAX_PROCESS];

void get_process_info(int selected_algo)
{
   printf("Enter total number of processes (maximum %d): ", MAX_PROCESS);
   scanf("%d", &num_of_process);
   if (num_of_process < 1)
   {
      exit(0);
   }

   for (int i = 0; i < num_of_process; i++)
   {
      printf("\nProcess %d:\n", i + 1);
      printf("==> Execution time: ");
      scanf("%d", &execution_time[i]);
      remain_time[i] = execution_time[i];
      if (selected_algo == 2)
      {
         printf("==> Deadline: ");
         scanf("%d", &deadline[i]);
      }
      else
```

```
        {
            printf("==> Period: ");
            scanf("%d", &period[i]);
        }
    }
}

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;
}

int get_observation_time(int selected_algo)
{
    if (selected_algo == 1)
    {
        return max(period[0], period[1], period[2]);
    }
    else if (selected_algo == 2)
    {
        return max(deadline[0], deadline[1], deadline[2]);
    }
}

void print_schedule(int process_list[], int cycles)
{
    printf("\nScheduling:\n\n");
    printf("Time: ");
    for (int i = 0; i < cycles; i++)
    {
        if (i < 10)
            printf("| 0%d ", i);
        else
            printf("| %d ", i);
    }
    printf("|\n");
    for (int i = 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");
    }
```

```
}

void rate_monotonic(int time)
{
   int process_list[100] = {0}, min = 999, next_process = 0;
   float utilization = 0;
   for (int i = 0; i < num_of_process; i++)
   {
      utilization += (1.0 * execution_time[i]) / period[i];
   }
   int n = num_of_process;
   int m = (float) (n * (pow(2, 1.0 / n) - 1));
   if (utilization > m)
   {
      printf("\nGiven problem is not schedulable under the said scheduling algorithm.\n");
   }
   for (int i = 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;
         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);
}

void earliest_deadline_first(int time){
   float utilization = 0;
   for (int i = 0; i < num_of_process; i++){
      utilization += (1.0*execution_time[i])/deadline[i];
   }
   int n = num_of_process;

   int process[num_of_process];
```

```
int max_deadline, current_process=0, min_deadline,process_list[time];
bool is_ready[num_of_process];

for(int i=0; i<num_of_process; i++){
   is_ready[i] = true;
   process[i] = i+1;
}

max_deadline=deadline[0];
for(int i=1; i<num_of_process; i++){
   if(deadline[i] > max_deadline)
      max_deadline = deadline[i];
}

for(int i=0; i<num_of_process; i++){
   for(int j=i+1; j<num_of_process; j++){
      if(deadline[j] < deadline[i]){
         int temp = execution_time[j];
         execution_time[j] = execution_time[i];
         execution_time[i] = temp;
         temp = deadline[j];
         deadline[j] = deadline[i];
         deadline[i] = temp;
         temp = process[j];
         process[j] = process[i];
         process[i] = temp;
      }
   }
}

for(int i=0; i<num_of_process; i++){
   remain_time[i] = execution_time[i];
   remain_deadline[i] = deadline[i];
}

for (int t = 0; t < time; t++){
   if(current_process != -1){
      --execution_time[current_process];
      process_list[t] = process[current_process];
   }
   else
      process_list[t] = 0;

   for(int i=0;i<num_of_process;i++){
      --deadline[i];
      if((execution_time[i] == 0) && is_ready[i]){
         deadline[i] += remain_deadline[i];
         is_ready[i] = false;
      }
      if((deadline[i] <= remain_deadline[i]) && (is_ready[i] == false)){
         execution_time[i] = remain_time[i];
         is_ready[i] = true;
      }
   }
```

```
        min_deadline = max_deadline;
        current_process = -1;
        for(int i=0;i<num_of_process;i++){
            if((deadline[i] <= min_deadline) && (execution_time[i] > 0)){
                current_process = i;
                min_deadline = deadline[i];
            }
        }
    }
    print_schedule(process_list, time);
}

void proportionalScheduling() {
    int n;
    printf("Enter the number of tasks: ");
    scanf("%d", &n);

    Task tasks[n];
    printf("Enter burst time and priority for each task:\n");
    for (int i = 0; i < n; i++) {
        tasks[i].id = i + 1;
        printf("Task %d - Burst Time: ", tasks[i].id);
        scanf("%d", &tasks[i].burst_time);
        printf("Task %d - Priority: ", tasks[i].id);
        scanf("%f", &tasks[i].priority);
    }

    // Sort tasks based on priority (ascending order)
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (tasks[j].priority > tasks[j + 1].priority) {
                // Swap tasks
                Task temp = tasks[j];
                tasks[j] = tasks[j + 1];
                tasks[j + 1] = temp;
            }
        }
    }

    printf("\nProportional Scheduling:\n");

    int total_burst_time = 0;
    float total_priority = 0.0;

    for (int i = 0; i < n; i++) {
        total_burst_time += tasks[i].burst_time;
        total_priority += tasks[i].priority;
    }

    for (int i = 0; i < n; i++) {
        float time_slice = (tasks[i].priority / total_priority) * total_burst_time;
        printf("Task %d executes for %.2f units of time\n", tasks[i].id, time_slice);
    }
}
```

```
int main()
{
   int option;
   int observation_time;

   while (1)
   {
   printf("\n1. Rate Monotonic\n2. Earliest Deadline first\n3. Proportional Scheduling\n\nEnter your choice:
");
   scanf("%d", &option);
   switch(option)
   {
      case 1: get_process_info(option);
            observation_time = get_observation_time(option);
            rate_monotonic(observation_time);
            break;
      case 2: get_process_info(option);
            observation_time = get_observation_time(option);
            earliest_deadline_first(observation_time);
            break;
      case 3: proportionalScheduling();
            break;
      case 4: exit (0);
      default: printf("\nInvalid Statement");
   }
   }
   return 0;
}
```

Output: (CPU Scheduling)

Rate Monotonic:

```
1. Rate Monotonic
2. Earliest Deadline first
3. Proportional Scheduling

Enter your choice: 1
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]: |    |    |####|####|    |    |    |    |    |    |    |    |####|####|    |    |    |    |    |    |
```

Earliest Deadline First:

```
1. Rate Monotonic
2. Earliest Deadline first
3. Proportional Scheduling

Enter your choice: 2
Enter total number of processes (maximum 10): 3

Process 1:
==> Execution time: 3
==> Deadline: 7

Process 2:
==> Execution time: 2
==> Deadline: 4

Process 3:
==> Execution time: 2
==> Deadline: 8

Scheduling:

Time: | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 |
P[1]: |    |    |####|####|####|    |    |    |
P[2]: |####|####|    |    |    |    |    |####|
P[3]: |    |    |    |    |    |####|####|    |
```

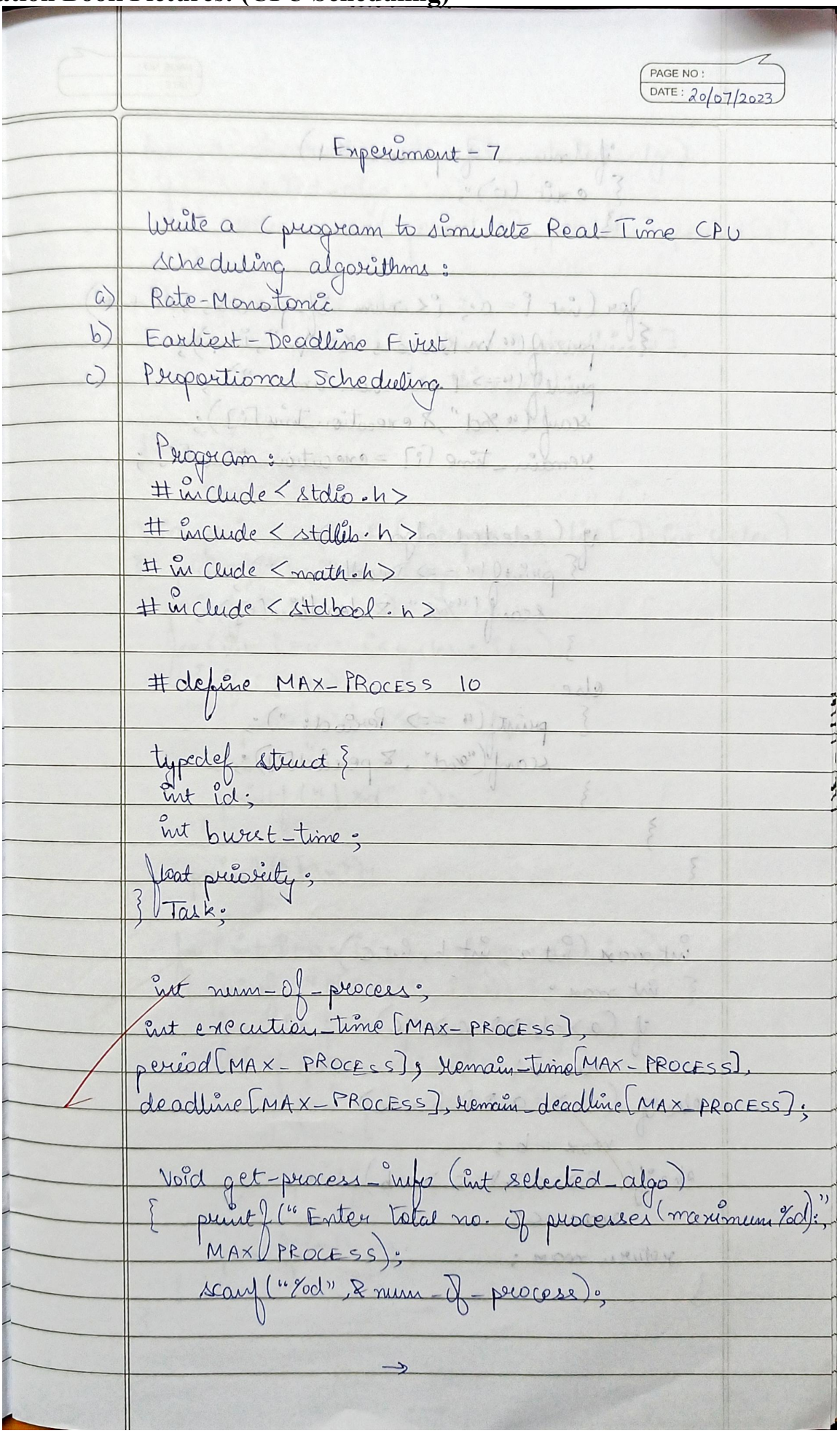
Proportional Scheduling:

```
1. Rate Monotonic
2. Earliest Deadline first
3. Proportional Scheduling

Enter your choice: 3
Enter the number of tasks: 3
Enter burst time and priority for each task:
Task 1 - Burst Time: 4
Task 1 - Priority: 2
Task 2 - Burst Time: 6
Task 2 - Priority: 3
Task 3 - Burst Time: 5
Task 3 - Priority: 1

Proportional Scheduling:
Task 3 executes for 2.50 units of time
Task 1 executes for 5.00 units of time
Task 2 executes for 7.50 units of time
```

Observation Book Pictures: (CPU Scheduling)



PAGE NO :
DATE : 20/07/2023

Experiment - 7

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

a) Rate-Monotonic
b) Earliest-Deadline First
c) Proportional Scheduling

Program :

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

#define MAX_PROCESS 10

typedef struct {
int id;
int burst_time;
float priority;
} Task;

int num_of_process;
int execution_time[MAX_PROCESS],
period[MAX_PROCESS], remain_time[MAX_PROCESS],
deadline[MAX_PROCESS], remain_deadline[MAX_PROCESS];

void get_process_info (int selected_algo)
{ printf("Enter total no. of processes (maximum %d):",
MAX_PROCESS);
scanf("%d", &num_of_process);
```

→

```
if (num_of_process < 1)
{ exit (0);
}

for (int i = 0; i < num_of_process; i++)
{
    printf("\n Process %d: \n", i+1);
    printf("==> Execution time : ");
    scanf("%d", &execution_time[i]);
    remain_time[i] = execution_time[i];

    if (selected_algo == 2)
    { printf("==> Deadline : ");
      scanf("%d", &deadline[i]);
    }
    else
    { printf("==> Period: ");
      scanf("%d", &period[i]);
    }
}
}

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;
}
```

```
int get_observation_time (int selected_algo)
{ if (selected_algo == 1)
    { return max (period[0], period[1], period[2]);
    }
  else if (selected_algo == 2)
    { return max (deadline[0], deadline[1],
                  deadline[2]);
    }
}

void print_schedule (int process_list[], int cycles)
{ printf ("\n Scheduling : \n\n");
  printf (" Time : ");
  for (int i=0 ; i<cycles ; i++)
    { if (i<10)
        printf ("| 0%d", i);
      else
        printf ("| %d", i);
    }
  printf ("|\n");

  for (int i=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");
    }
}
```

→

DATE :

```
Void rate_monotonic (int time)
{ int process_list[100] = {0}, min = 999,
      next_process = 0;
  float utilization = 0;

  for (int i=0; i< num_of_process; i++)
    { utilization += (1.0 * execution_time[i])/period[i];
    }

  int n = num_of_process;
  int m = (float)(n*(pow(2, 1.0/n)-1));

  if (utilization > m)
  { printf("\n Given problem is not schedulable
                under the said scheduling algorithm.\n");
  }

  for (int i = 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;
      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);
}

void earliest_deadline_first(int time)
{ float utilization = 0;
    for (int i = 0; i < num_of_process; i++)
    { utilization += (1.0 * execution_time[i] /
                      deadline[i]);
    }
    int n = num_of_process;
    if (utilization > 1)
    { printf("Given problem is not schedulable
              under the said algorithm\n");
    }

    int process[num_of_process];
    int max_deadline, current_process = 0,
        min_deadline, process_list[time];

    bool is_ready[num_of_process];

    for (int i = 0; i < num_of_process; i++)
    { is_ready[i] = true;
      process[i] = i+1;
    }
```

→

```
max_deadline = deadline[0];
for(int i=1; i<num_of_process; i++)
{ if(deadline[i] > max_deadline)
    max_deadline = deadline[i];
}

for(int i=0; i<num_of_process; i++)
{ for(j=j+1; j<num_of_process; j++)
  { if(deadline[j]<deadline[i])
    { int temp = execution_time[j];
      execution_time[j]= execution_time[i];
      execution_time[i] = temp;
      temp = deadline[j];
      deadline[j] = deadline[i];
      deadline[i] = temp;
      temp = process[j];
      process[j] = process[i];
      process[i] = temp;
    }
  }
}

for(int i=0; i<num_of_process; i++)
{ remain_time[i] = execution_time[i];
  remain_deadline[i] = deadline[i];
}

for(int t=0; t<time; t++)
{ if( current_process != -1)
  { --execution_time[current_process];
    process_list[t] = process[current_process];
  }
```

```
else
    process_list[t] = 0;

for(int i=0; i<num_of_process; i++)
{   --deadline[i];
    if((execution_time[i]==0 && is_ready[i])
    {  deadline[i] += remain_deadline[i];
       is_ready[i] = false;
    }

    if((deadline[i] <= remain_deadline[i])) &&
       (is_ready[i] == false))
    {  execution_time[i] = remain_time[i];
       is_ready[i] = true;
    }
}

min_deadline = max_deadline;
current_process = -1;
for(int i=0; i<num_of_process; i++)
{  if((deadline[i] <= min_deadline ) &&
      (execution_time[i] > 0))
   {  current_process = i;
      min_deadline = deadline[i];
   }
}
}
print_schedule(process_list, time);
}
```

→

```
void proportional Scheduling ()
{ int n;
  printf (" Enter the no. of tasks : ");
  scanf (" %d", &n);

  Task tasks [n];
  printf (" Enter the burst time and priority for
          each task: \n");
  for (int i=0; i<n; i++)
  { tasks [i]. id = i+1;
    printf (" Task %d - Burst Time : ", tasks[i].id);
    scanf ("%d", &tasks[i]. burst_time);
    scanf ("%f", &tasks [i]. priority);
  }

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

  printf ("\n Proportional Scheduling : \n");
  int burst_time = 0;
  int total priority = 0.0;

  for ( int i=0; i<n; i++)
  { total_burst_time += tasks [i]. burst_time;
    total_priority += tasks [i]. priority;
  }
```

```
        for(int i=0; i<n; i++)
        {   float time_slice = (tasks[i].priority/
                                total_priority) *
                                total_burst_time;
            printf("Task %d executes for %.2f units
                of time\n", tasks[i].id, time_slice);
        }
    }

int main()
{ int choice;
  int observation_time;
  printf("\n1. Rate Monotonic\n2. Earliest
        Deadline First\n3. Proportional
        Scheduling\n Enter your choice: ");
  scanf("%d", &choice);

  switch(choice)
  { case 1 : get_process_info(choice);
             observation_time = get_observation_time
                                (choice);
             rate_monotonic(observation_time);
             break;

    case 2 : get process_info(choice);
             observation_time = get_observation_time
                                (choice);
             earliest_deadline_first(observation_
                                time);
             break;

    case 3 : proportionalScheduling();
             break;
```

```
        default: printf("\n Invalid Statement");
    }

    return 0;
}
```

Output

1. Rate Monotonic
2. Earliest Deadline First
3. Proportional Scheduling.

Enter your choice : 1

Enter total no. of processes (max 10) : 3

Process 1 :

=> Execution time : 3

=> Period : 20

Process 2 :

=> Execution time : 2

=> Period : 5

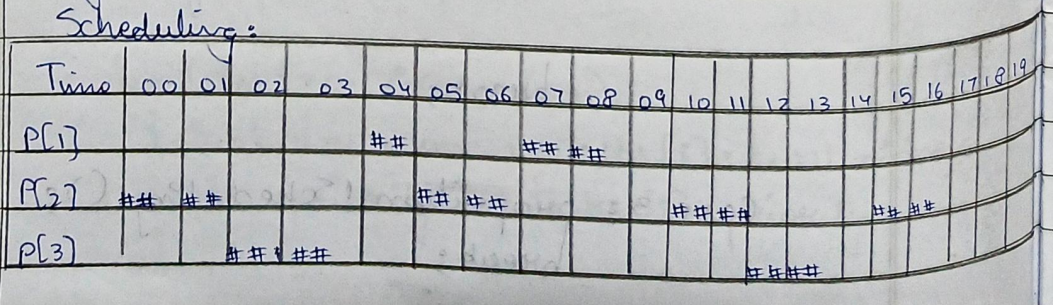
Process 3 :

=> Execution time : 2

=> Period : 10

Given problem is schedulable under the said algorithm.

Scheduling :



PAGE NO :
DATE :

1. Rate Monotonic
2. Earliest Deadline first
3. Proportional Scheduling

Enter your choice : 2

Enter total no. of processes (max 10) : 3

Process 1 :
=> Execution time : 3
=> Deadline : 7

Process 2 :
=> Execution time : 2
=> Deadline : 4

Process 3 :
=> Execution time : 2
=> Deadline : 8

Scheduling :

Time	00	01	02	03	04	05	06	07
P[1]			##	##	##			
P[2]	##	##						##
P[3]						##	##	

→

1. Rate Monotonic
2. Earliest Deadline First
3. Proportional Scheduling.

Enter your choice: 3

Enter no. of tasks: 3

Enter burst time & priority for each task:
Task-1 - Burst Time: 4
Task-1 - Priority: 2

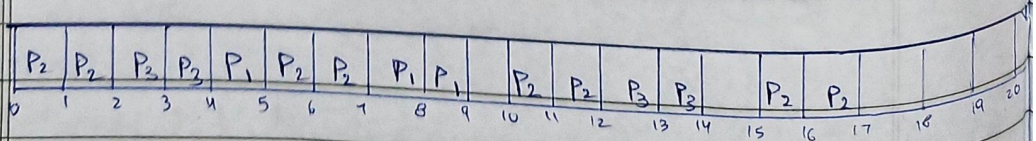
Task-2 - Burst Time: 6
Task-2 - Priority: 3

Task-3 - Burst Time: 5
Task-3 - Priority: 1

Proportional Scheduling:
Task 3 executes for 2.50 units of time
Task 1 executes for 5.00 units of time
Task 2 executes for 7.50 units of time.

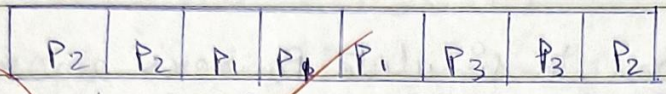
Traversal:

1. Rate Monotonic:



PAGE NO :
DATE :

2. Earliest Deadline First.



9/10

22/7/23

2) Producer-Consumer Problem

Program:

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

int mutex=1,full=0,empty=3,x=0;

int main()
{
   int n;
   void producer();
   void consumer();
   int wait(int);
   int signal(int);
   printf("\n1.Producer\n2.Consumer\n3.Exit");
   while(1)
   {
      printf("\nEnter your choice: ");
      scanf("%d&",&n);
      switch(n)
      {
         case 1: if((mutex==1)&&(empty!=0))
                 producer();
                 else
                 printf("Buffer is full!!");
                 break;
         case 2: if((mutex==1)&&(full!=0))
                 consumer();
                 else
                 printf("Buffer is empty!!");
                 break;
         case 3: exit(0);
                 break;
      }
   }
   return 0;
}

int wait(int s)
{
   return (--s);
}

int signal(int s)
{
   return(++s);
}

void producer()
{
   mutex=wait(mutex);
   full=signal(full);
   empty=wait(empty);
   x++;
   printf("\nProducer produces the item %d",x);
```

```
    mutex=signal(mutex);
}

void consumer()
{
    mutex=wait(mutex);
    full=wait(full);
    empty=signal(empty);

    printf("\nConsumer consumes item %d",x);
    x--;
    mutex=signal(mutex);
}
```

Output: (Producer - Consumer Problem)

```
1.Producer
2.Consumer
3.Exit
Enter your choice: 1

Producer produces the item 1
Enter your choice: 2

Consumer consumes item 1
Enter your choice: 2
Buffer is empty!!
Enter your choice: 1

Producer produces the item 1
Enter your choice: 1

Producer produces the item 2
Enter your choice: 1

Producer produces the item 3
Enter your choice: 1
Buffer is full!!
Enter your choice: 3
```

Observation Book Pictures: (Producer - Consumer Problem)

PAGE NO :
DATE : 20/7/2023

Experiment - 5

Write a C program to simulate producer-Consumer problem using semaphores.

Program:

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

int mutex=1, full=0, empty=3, x=0;

int main()
{ int n;
  void producer();
  void consumer();
  int wait(int);
  int signal(int);
  printf("\n1. Producer\n2. Consumer\n3. Exit");
  while(1)
  { printf("\nEnter your choice: ");
    scanf("%d",&n);
    switch(n)
    { case 1: if((mutex==1)&&(empty!=0))
                producer();
              else
                printf("Buffer is full!!");
              break;

      case 2: if((mutex==1)&&(full!=0))
                consumer();
              else
                printf("Buffer is Empty!!");
              break;
```

```
        case 3: exit(0);
                break;
        }
    }
    return 0;
}

int wait(int s)
{ return (--s);
}

int signal (int s)
{ return(++s);
}

void producer()
{ mutex = wait(mutex);
  full = signal(full);
  empty = wait(empty);
  x++;
  printf("\nProducer produces the item %d", x);
  mutex = signal(mutex);
}

Void consumer()
{ mutex = wait(mutex);
  full = wait(full);
  empty = signal(empty);
  printf("Consumer consumes item %d",x);
  x--;
  mutex = signal(mutex);
}
```

Output:

1. Producer
2. Consumer
3. Exit

Enter your choice : 1

Producer produces item 1

Enter your choice : 2

Consumer consumes item 1

Enter your choice : 2

Buffer is Empty!!

Enter your choice : 1

Producer produces item 1

10/10

Enter your choice : 1

producer produces item 2

27/7/23

Enter your choice : 1

Producer produces item 3.

Enter your choice : 1

Buffer is full!!

Enter your choice : 3.

3) Dining - Philosopher Problem:

Program:

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

#define N 5
#define THINKING 2
#define HUNGRY 1
#define EATING 0
#define LEFT (num_of_philosopher + 4) % N
#define RIGHT (num_of_philosopher + 1) % N

int state[N];
int phil[N] = {0,1,2,3,4};

sem_t mutex;
sem_t S[N];

void test(int num_of_philosopher)
{
		if (state[num_of_philosopher] == HUNGRY && state[LEFT] != EATING &&
state[RIGHT] != EATING)
		{
			state[num_of_philosopher] = EATING;

			sleep(2);

			printf("Philosopher %d takes fork %d and %d\n", num_of_philosopher +1, LEFT +1,
num_of_philosopher +1);

			printf("Philosopher %d is Eating\n", num_of_philosopher +1);

			sem_post(&S[num_of_philosopher]);
	}
}

void take_fork(int num_of_philosopher)
{

	sem_wait(&mutex);

	state[num_of_philosopher] = HUNGRY;

	printf("Philosopher %d is Hungry\n", num_of_philosopher +1);

	test(num_of_philosopher);

	sem_post(&mutex);

	sem_wait(&S[num_of_philosopher]);

	sleep(1);
}
```

```
void put_fork(int num_of_philosopher)
{

	sem_wait(&mutex);

	state[num_of_philosopher] = THINKING;

	printf("Philosopher %d putting fork %d and %d down\n",num_of_philosopher +1, LEFT +1,
num_of_philosopher +1);
	printf("Philosopher %d is thinking\n", num_of_philosopher +1);

	test(LEFT);
	test(RIGHT);

	sem_post(&mutex);
}

void* philosopher(void* num)
{

	while (1)
	{
		int* i = num;
		sleep(1);
		take_fork(*i);
		sleep(0);
		put_fork(*i);
	}
}

int main()
{
	int i;
	pthread_t thread_id[N];

	sem_init(&mutex,0,1);

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

	sem_init(&S[i],0,0);

	for (i =0; i < N; i++) {

	pthread_create(&thread_id[i],NULL,philosopher, &phil[i]);

	printf("Philosopher %d is thinking\n", i +1);
   }

	for (i =0; i < N; i++)
	{
		pthread_join(thread_id[i],NULL);
	}
}
```

Output: (Dining - Philosopher Problem)

```
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 1 is Hungry
Philosopher 5 is Hungry
Philosopher 4 is Hungry
Philosopher 3 is Hungry
Philosopher 2 is Hungry
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eating
Philosopher 2 putting fork 1 and 2 down
Philosopher 2 is thinking
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 2 is Hungry
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is thinking
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eating
Philosopher 1 is Hungry
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinking
Philosopher 4 takes fork 3 and 4
Philosopher 4 is Eating
Philosopher 3 is Hungry
Philosopher 2 putting fork 1 and 2 down
Philosopher 2 is thinking
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
```

Observation Book Pictures: (Dining - Philosopher Problem)

PAGE NO :
DATE : 20/07/2023

Experiment - 6

Write a C program to simulate the concept of Dining - Philosophers problem.

Program :

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

#define N 3
#define THINKING
#define HUNGRY
#define EATING
#define LEFT (num_of_philosopher+4) % N
#define RIGHT (num_of_philosopher + 1) % N

int state[N];
int phil[N] = {0,1,2,3,4};

sem_t mutex;
sem_t S[N];

void test (int num_of_philosopher)
{ if (state[num_of_philosopher] == HUNGRY &&
    state[LEFT] != EATING &&
    state[RIGHT] != EATING)
  { state[num_of_philosopher] = EATING;
    sleep(2);
    printf("Philosopher %d takes fork %d and %d\n",
           num_of_philosopher+1, LEFT+1,
           num_of_philosopher+1);
    printf("Philosopher %d is Eating\n",
           num_of_philosopher+1);
```

```
            sem_post (&S[num_of_philosopher]);
        }
    }

Void take_fork (int num_of_philosopher)
{   sem_wait (&mutex);

    state [num_of_philosopher] = HUNGRY;
    printf("Philosopher %d is Hungry\n", num_of_philosopher+1);

    test (num_of_philosopher);
    sem_post(&mutex);
    sem_wait (&S[num_of_philosopher]);
    sleep(1);
}

Void put_fork (int num_of_philosopher)
{   sem_wait (&mutex);
    state [num_of_philosopher] = THINKING;

    printf("Philosopher %d putting %d and %d down\n",
            num_of_philosopher+1, LEFT+1,
            num_of_philosopher+1);

    test (LEFT);
    test (RIGHT);
    sem_post (&mutex);
}

void * philosopher (void* num)
{   while(1)
    {   int *i = num;
        sleep (1);
```

```
        take_fork(*i);
        sleep(0);
        put_fork(*i);
    }
}

int main()
{ int i;
  pthread_t thread_id[N];

  sem_init (&mutex, 0, 1);
  for (i=0; i< N; i++)
     sem_init (&S[i], 0, 0);
  for ( i=0; i<N; i++)
  { pthread_create (&thread_id[i], NULL,
                    philosopher, &phil[i]);
    printf(" Philosopher %d is thinking \n", i+1);
  }

  for (i=0; i< N; i++)
  { pthread_join (thread_id[i], NULL);
  }
}
```

Output:

```
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 1 is Hungry
Philosopher 3 is Hungry
Philosopher 3 takes fork 1 and 3
Philosopher 3 is eating
Philosopher 2 is Hungry
```

Philosopher 3 putting fork 1 and 3 down
Philosopher 3 is thinking
Philosopher 1 takes fork 2 and 1
Philosopher 1 is eating
Philosopher 3 is Hungry
Philosopher 1 putting fork 2 and 1 down
Philosopher 1 is thinking
Philosopher 2 takes fork 3 and 2
Philosopher 2 is eating
Philosopher 1 is Hungry
Philosopher 2 putting ~~down~~ fork 3 and 2 down
Philosopher 2 is thinking
Philosopher 3 takes fork 1 and 3
Philosopher 3 is eating
Philosopher 2 is Hungry
Philosopher 3 putting fork 1 and 3 down