



Ecole Centrale de Nantes

M2 - CORO-Embedded Real-Time Systems

Partitioning Strategies using Rate Monotonic Scheduling

Author:

Ragesh Ramachandran

Supervisor:

Maryline Chetto

December 12, 2018

Contents

1	Introduction	2
1.1	Objective	3
1.2	Theory	3
1.2.1	Greedy Approximation Algorithm	3
2	Program Architecture	5
3	Program Development	7
3.1	Data acquisition	7
3.2	Random Task generation(Optional)	9
3.3	Feasibility Analysis	10
3.4	Partitioning	10
3.4.1	Rate Monotonic Next Fit Partitioning	11
3.4.2	Rate Monotonic First Fit Partitioning	12
3.4.3	Rate Monotonic Best Fit Partitioning	14
3.5	Display of Metrics	15
4	Results	17
4.1	Case 1 - User Input	17
4.2	Case 2 - Random task generation	22
4.3	Case 3 - Random task generation	27
5	Conclusion	31
6	Appendix	32

Chapter 1

Introduction

A real-time system is a system that must produce logically correct results in the specified time. All the tasks submitted to the system have known timing requirements and are called as real-time tasks. The processing of each task must be completed by the end of a task's period, called as the deadline of the task instance. The requirements of a periodic real-time task τ_i are characterized by a period T_i and a worst-case computation time C_i . The utilization factor of a task is defined to be $\frac{C_i}{T_i}$.

A real-time system must ensure that each task instance will complete before its deadline. This is done by an admission control and a scheduling policy for the real-time system. The admission control is an algorithm that ensures that only tasks that will meet their deadlines are accepted in the system. These tasks that does not miss their deadlines are called as schedulable tasks. One of the most widely used scheduling policies for preemptive periodic real-time tasks is rate-monotonic scheduling in which the priority of tasks are based on their period.

Since multiprocessor systems are becoming more common for real-time applications scheduling real-time tasks on multiprocessor systems is also an important problem. Tasks in a multiprocessor systems can be scheduled by global scheduling or partitioning. In a global scheduling scheme, tasks can execute on any processor and, after being preempted, can be resumed on a different processor. In a partitioning scheme, each task is assigned to a processor and is only executed on this processor.

The task scheduling problem in a multiprocessor systems consists of two sub problems: task allocation to individual processors and task scheduling at the individual processors. The task allocation problem is an NP hard problem. A decision problem H is NP-hard when for every problem L in NP, there is a polynomial-time reduction from L to H.

1.1 Objective

The objective of this lab is to simulate the behaviour of several partitioning strategies in periodic tasks with their deadlines equal to their periods. Each processor is scheduled using Rate Monotonic Scheduler. The program will be composed of the following parts:

- **Data acquisition:** The user will specify the number of tasks and for each task, its WCET and period. We will assume that the task set is synchronous to time zero.
- **Partitioning:** This part will focus on the partitioning of the tasks. Simulate BF (Best Fit), FF (First Fit) and NF (Next Fit). Each strategy will be coded as a function which input is a set of tasks and the outputs are for each processor, the processor utilization and identity of the tasks assigned to it.
- **Display of metrics:** The number of processors used, highest processor utilization, lowest processor utilization of the task sets are calculated.

1.2 Theory

Multiprocessor scheduling problem can be seen as a bin packing problem, tasks of different utilization factors must be allocated into a finite number of processors. In computational complexity theory this problem is a combinatorial NP hard problem. The decision problem of deciding if the tasks fit into a specified number of processors is NP-complete.

In the bin packing problem we are given a set of n tasks $\{1, 2, \dots, n\}$ and each task has a utilization factor $U_i \leq 1$. The goal is to find the minimum number of processors with a capacity 1 into which these tasks can be allocated.

1.2.1 Greedy Approximation Algorithm

The greedy algorithm can be used for bin packing problem. In greedy bin packing algorithm, a new processor is added only if the task can not be scheduled in any of the already available processors. However, there might be several available processors in which the task i can be scheduled. The bin-packing algorithm assigns tasks to processors and uses the schedulability bound to determine if a processor can accept a task. The tasks are ordered according to their period, priority or processor utilization and the task is assigned in their order. For each task, the processor is made available according to a policy.

Algorithm 1 Greedy approximation algorithm

```
1: procedure Greedy algorithm
2:   Input:  $\tau_1, \tau_2 \dots \tau_n$ 
3:   Sort the task sets  $\tau_1, \tau_2 \dots \tau_n$  according to period, priority or utilization
4:    $j = 0$ 
5:   for  $i=1$  to  $n$  do
6:     Determine the utilization of each task
7:     if Task  $i$  fits into processor  $j$  then
8:       Fit  $i$  into  $j$ 
9:     else
10:       $j = j + 1$ 
11:      Fit  $i$  into  $j$ 
```

- **Next Fit:** When processing the next task, see if it fits in the same processor of the last task. Use a new processor only if it does not fit.
- **First Fit:** Rather than checking just the last processor, we check all previous processors to see if the next task will fit. Start a new processor, only when it does not.
- **Best Fit:** Fit task to the processor that would have least amount of space left after the allocation of task. This places the next task in the tightest spot.

Chapter 2

Program Architecture

The program is implemented in python3 and the following functions are created for performing the partitioning algorithm.

- The data structure *Class task* to store the data of each task with the following attributes

```
1 struct task
2 {
3     int task_id
4     int period
5     int WCET
6     float U
7 }
```

- The data structure *Class core* to store the data of each processor with the following attributes

```
1 struct core
2 {
3     int core_id
4     float core_U
5     float core_rem_U
6 }
```

- A function *read_data()* to read the data from the user and store it in the task data structure.
- A function *random_data()* to generate random number of tasks and task sets for the statistical evaluation of the three partitioning strategies
- A function *schedulability()* to check the schedulability of each task based on Rate Monotonic scheduling.

- A function *truncate(f, n)* to truncate the float result *f* to *n* decimal places without rounding off.
- A function *hyperperiod()* to calculate the hyperperiod of the task sets.
- A function *NEXT_FIT()* to implement the next fit partitioning algorithm on the given task sets.
- A function *FIRST_FIT()* to implement the first fit partitioning algorithm on the given task sets.
- A function *BEST_FIT()* to implement the best fit partitioning algorithm on the given task sets.
- A function *display_metrics* to display the metrics after partitioning in three different strategies.
- A function *main()* that calls all the above mentioned functions in a sequential order.

```
1 main()  
2 {  
3     read_data()  
4     random_data()  
5     hp = hyperperiod()  
6     schedulability()  
7     NEXT_FIT()  
8     FIRST_FIT()  
9     BEST_FIT()  
10 }
```

Chapter 3

Program Development

The program for the partitioning of the tasks among the processors were implemented in **python3**. The task sets given in table 4.3 is used for the evaluation the developed program.

	Period	WCET
Task 1	8	4
Task 2	8	2
Task 3	8	4
Task 4	10	2

Table 3.1: Test task sets

3.1 Data acquisition

The number of tasks to be scheduled (n) and then the worst case execution time ($WCET$) and Time period ($Period$) is read from the user and then stored in a class data structure as given in 3.1. The data acquisition is performed in the function `read_data()`. The function calculates the utilization factor online and then the tasks are sorted in the increasing order of their period. The output is displayed in the console for the user.

The utilization factor of each task is calculated using the given equation where n is the number of tasks, C_i is the worst case execution time and T_i is the period of the task i .

$$U = \sum_{i=1}^n C_i/T_i$$

Listing 3.1: Data acquisition

```

1  class task:
2      def __init__(self, task_id = None, period = None, WCET = None, U = None):
3          self.task_id = task_id
4          self.period = period
5          self.WCET = WCET
6          self.U = U
7

```

Listing 3.2: Data acquisition

```

1
2  def read_data():
3      global n
4      global tasks
5      tasks = []
6      n = int(input("\n \t\tEnter number of tasks:"))
7      # Read data from user
8      for i in range(n):
9          task_id = i
10         print("\nEnter Period of task T", i, ":")
11         period = int(input())
12         print("Enter the WCET of task C", i, ":")
13         WCET = int(input())
14         u = WCET/period
15         #Truncating the float result to 2 decimal places
16         U = truncate(u, 2)
17         tasks.append(task(task_id, period, WCET, U))
18
19     # Tasks are sorted based on their period
20     tasks = sorted(tasks, key=lambda tasks: tasks.period)
21     for i in range(n):
22         print("_____")
23         print("TASK   %d"%(i+1))
24         print("_____")
25         print("Period       ", tasks[i].period)
26         print("WCET         ", tasks[i].WCET)
27         print("Utilization", tasks[i].U)
28         print("_____")
29     print("\n\n")

```

3.2 Random Task generation(Optional)

A function *random_data()* was created which randomly generates the number of tasks which is needed to be partitioned. Then the period and WCET of the tasks are also randomly generated after which the utilization factor of each task set is computed. The output of the function is the set of tasks which is then passed into the partitioning function where it is partitioned according to next fit, first fit and best fit algorithms. This was created in order to get a statistical evaluation of the three partitioning strategies.

Listing 3.3: Random task set generation

```
1 def random_data():
2     global n      # Number of tasks to be partitioned
3     global tasks  # List that stores the instances of tasks
4     tasks = []
5     random.seed()
6     n = random.randrange(2,7)
7     print("number of tasks",n)
8
9     for i in range(n):
10        task_id = i
11        period = random.randrange(9,20)
12        WCET = random.randrange(4,8)
13        u = WCET/period
14        # Limit U by 2 decimal places without rounding off
15        U = truncate(u,2)
16        tasks.append(task(task_id,period,WCET,U))
17
18    # Tasks are sorted based on their period and displayed
19    tasks = sorted(tasks, key=lambda tasks:tasks.period)
20    for i in range(n):
21        print("_____")
22        print("TASK   %d"%(i+1))
23        print("_____")
24        print("Period      ",tasks[i].period)
25        print("WCET        ",tasks[i].WCET)
26        print("Utilization",tasks[i].U)
27        print("_____")
28        print("\n\n")
```

3.3 Feasibility Analysis

The tasks are allocated processors based on the feasibility test of RM scheduling. The feasibility analysis is performed on the tasks using the schedulability equation 3.1 which is a sufficient condition, but not necessary. Another schedulability test is also performed which is given in equation 3.2. This is a sufficient and necessary condition for Rate Monotonic scheduling.

$$U \leq n(2^{1/n} - 1) \quad (3.1)$$

$$U \leq 1, \forall T_j \text{ multiple of } (T_{j+1}) \quad (3.2)$$

Where n is the number of tasks. The feasibility test is implemented inside the function *schedulability()*

Listing 3.4: Schedulability test

```
1 def schedulability():
2     global sched_factor
3     # Schedulability test for RM scheduling
4     sched_fac = n*(2**(1/n)-1)
5     # to limit only 2 decimal places without rounding off
6     sched_factor = truncate(sched_fac, 2)
7     print("\tsched_factor", sched_factor)
```

3.4 Partitioning

For the partitioning of task among the processors a data structure is created called *core* given in 3.5. The *class core* contains the following parameters:

- **core_id**: The current core number in process.
- **core_U**: The total utilization factor of load.
- **core_rem_U**: The remaining utilization after task allocation.
- **task_ID**: The name of the allocated task.
- **task_U**: The utilization of the allocated task.

Listing 3.5: Schedulability test

```

1 class core:
2     def __init__(self, core_id=0, core_U=0, core_rem_U=0, task_ID=0, task_U=0):
3         self.core_id    = core_id
4         self.core_U     = core_U
5         self.core_rem_U = core_rem_U
6         self.task_ID    = task_ID
7         self.task_U     = task_U

```

3.4.1 Rate Monotonic Next Fit Partitioning

- initialize $task\ i = 1$ and $processor\ m = 1$.
- Assign $task\ i$ on $processor\ m$ if the feasibility tests is true.
- Else, put $task\ i$ on $processor\ m + 1$.
- Assign the next $task\ i + 1$.
- Stop when all tasks have been assigned.
- m is the number of required processors

Listing 3.6: Next Fit Partitioning

```

1 def NEXT_FIT():
2     # 'n' tasks and 'm' processors
3     m = 1                #core counter
4     c = 1                #initial core count
5     cores = []           #instance list of core class
6     core_rem = sched_factor #Store remaining U factor of core
7     core_buff = []        #Store the utilization of cores
8     merged_cores = []     #Store the non-duplicate cores
9     id_list = []          #Stores id list of cores
10    id_merged = []         #Stores id list of merged cores
11    for i in range(n):
12        # if a task cannot fit into same core
13        if(tasks[i].U > core_rem): #task does not fit into same core
14            c = c + 1             #task i is added to a new core
15            core_rem = sched_factor - tasks[i].U #calculate rem capacity
16            core_rem = truncate(core_rem,2)
17            cores.append(core(c, sched_factor, core_rem,i,tasks[i].U))
18            # If a task fit into a same core
19    else:

```

```

20     core_rem = core_rem - tasks[i].U #task run on same core
21     core_rem = truncate(core_rem,2)
22     cores.append(core(c, sched_factor, core_rem,i,tasks[i].U))
23 # if there are no new cores added then default value = 1
24 if(c>m):
25     m = c
26 # sorting the processors based on core_id
27 cores = sorted(cores, key=lambda cores:cores.core_id )
28 for i in range(len(cores)):
29     id_list.append(cores[i].core_id)
30     print("_____")
31     print("\nCORE  %d"%cores[i].core_id)
32     print("\tcore number      ",cores[i].core_id)
33     print("\tcore load capacity",cores[i].core_U)
34     print("\tcore rem capacity ",cores[i].core_rem_U)
35     print("\tTask in core      ",cores[i].task_ID)
36     print("\tTask Utilization  ",cores[i].task_U)
37     print("_____")

```

3.4.2 Rate Monotonic First Fit Partitioning

- initialize *task i = 1* and *processor res = .*
- Assign *task i* on *processor res* if the feasibility tests is true.
- Else, Loop over all the available processors and check first processor for feasibility.
- If the feasibility test is true, then assign *task i+1* on *processor res*
- If feasibility test is false,then assign *task i+1* on *processor res+1*
- Stop when all tasks have been assigned.
- *res* is the number of required processors

Listing 3.7: First Fit Partitioning

```

1 def FIRST_FIT():
2     # 'n' tasks and 'res' processors
3     global need
4     res = 0
5     core_remain = [0]*n      # array to store remaining space in cores
6     cores = []              #instance list of core class
7     core_rem = sched_factor #Store remaining U factor of core

```

```

8  core_buff = []           #Store the utilization of cores
9  merged_cores = []       #Store the non-duplicate cores
10 id_list = []            #Stores id list of cores
11 id_merged = []          #Stores id list of merged cores
12
13 for i in range(n):
14     need = 0
15     # Find the first core that can schedule the task
16     for j in range(res):
17         if core_remain[j] >= tasks[i].U:
18             core_remain[j] = tasks[i].U - core_remain[j]
19             cores.append(core(res,
20                             sched_factor,
21                             truncate(tasks[i].U - core_remain[j], 2),
22                             i,
23                             tasks[i].U))
24             break
25         else:
26             need = need + 1
27
28     # If no core can schedule the task add new core
29     if need == res:
30         core_remain[res] = sched_factor - tasks[i].U
31         res = res + 1 #task i is added to a new core
32         cores.append(core(res,
33                             sched_factor,
34                             truncate(sched_factor - tasks[i].U, 2),
35                             i,
36                             tasks[i].U))
37
38 for i in range(len(cores)):
39     print("_____")
40     print("\nCORE   %d"%cores[i].core_id)
41     print("\tcore number       ",cores[i].core_id)
42     print("\tcore load capacity",cores[i].core_U)
43     print("\tcore rem capacity ",cores[i].core_rem_U)
44     print("\tTask in core       ",cores[i].task_ID)
45     print("\tTask Utilization  ",cores[i].task_U)
46     print("_____")
47 print("\n\tNumber of processors used for FIRST FIT",res)

```

3.4.3 Rate Monotonic Best Fit Partitioning

- initialize *task i = 1* and *processor res = 1*.
- Assign *task i* on *processor res* if the feasibility tests is true.
- Else, Loop over all the available processors and check for processor with least space and test feasibility.
- If the feasibility test is true, then assign *task i+1* on *processor res*
- If feasibility test is false, then assign *task i+1* on *processor res+1*
- Stop when all tasks have been assigned.
- *res* is the number of required processors

Listing 3.8: Best Fit Partitioning

```
1 def BEST_FIT():
2     # 'n' tasks and 'm' processors
3     res = 0
4     core_remain = [0]*n # array to store remaining space in cores
5     cores = []          #instance list of core class
6     core_rem = sched_factor #Store remaining U factor of core
7     core_buff = []      #Store the utilization of cores
8     merged_cores = []   #Store the non-duplicate cores
9     id_list = []        #Stores id list of cores
10    id_merged = []       #Stores id list of merged cores
11    count = sched_factor
12
13    for i in range(n):
14        min = count + 1
15        for j in range(res):
16            if core_remain[j] >= tasks[i].U and core_remain[j]-tasks[i].U < min:
17                min = core_remain[j] - tasks[i].U
18                c = j
19                break
20
21        if min == count+1:
22            core_remain[res] = sched_factor - tasks[i].U
23            res = res + 1
24            cores.append(core(res,
25                             sched_factor,
26                             truncate(sched_factor - tasks[i].U,2),
```

```

27         i,
28         tasks[i].U))
29     else:
30         core_remain[c] = tasks[i].U - core_remain[c]
31         cores.append(core(res,
32                         sched_factor,
33                         truncate(tasks[i].U - core_remain[c],2),
34                         i,
35                         tasks[i].U))
36
37     for i in range(len(cores)):
38         print("_____")
39         print("\nCORE  %d"%cores[i].core_id)
40         print("\tcore number      ",cores[i].core_id)
41         print("\tcore load capacity",cores[i].core_U)
42         print("\tcore rem capacity ",cores[i].core_rem_U)
43         print("\tTask in core      ",cores[i].task_ID)
44         print("\tTask Utilization  ",cores[i].task_U)
45         print("_____")
46     print("\n\tNumber of processors used for BEST FIT",res)

```

3.5 Display of Metrics

The number of processors used by each partitioning strategies are calculated and displayed. The maximum and minimum utilization factor of the the available processors are also calculated in the program.

Listing 3.9: Display of Metrics

```

1  # sorting the processors based on core_id
2  cores = sorted(cores, key=lambda cores:cores.core_id )
3  for i in range(len(cores)):
4      id_list.append(cores[i].core_id)
5
6  # Metrics are calculated here
7  # Inorder to remove the multiple instances of
8  # core with same core_id they are merged
9  id_merged = [i for i, x in enumerate(id_list)
10              if i == len(id_list) - 1 or x != id_list[i + 1]]
11  for i in id_merged:
12      merged_cores.append(cores[i])
13
14  # Display of main metrics

```



```
15  for i in range(len(merged_cores)): #truncate the utilization value
16      core_buff.append(truncate(merged_cores[i].core_U
17                          - merged_cores[i].core_rem_U,2))
18  core_buff.sort() #sorting the U list for finding max and min values
19  print("Utilization factor of cores      ", core_buff)
20  print("Maximum Utilization factor of cores", core_buff[-1])
21  print("Minimum Utilization factor of cores", core_buff[0])
```

Chapter 4

Results

4.1 Case 1 - User Input

	Period	WCET
Task 1	8	4
Task 2	8	2
Task 3	8	4
Task 4	10	2

Table 4.1: Test task 1

Listing 4.1: Data acquisition

```
1      Enter number of tasks:4
2
3      Enter Period of task T 0 :
4      8
5      Enter the WCET of task C 0 :
6      4
7
8      Enter Period of task T 1 :
9      8
10     Enter the WCET of task C 1 :
11     2
12
13     Enter Period of task T 2 :
14     8
15     Enter the WCET of task C 2 :
16     4
```

```

17
18   Enter Period of task T 3 :
19   10
20   Enter the WCET of task C 3 :
21   2
22
23   _____
24   TASK  1
25   _____
26   Period      8
27   WCET        4
28   Utilization 0.5
29   _____
30   _____
31   TASK  2
32   _____
33   Period      8
34   WCET        2
35   Utilization 0.25
36   _____
37   _____
38   TASK  3
39   _____
40   Period      8
41   WCET        4
42   Utilization 0.5
43   _____
44   _____
45   TASK  4
46   _____
47   Period      10
48   WCET        2
49   Utilization 0.2
50   _____
51
52

```

Listing 4.2: Metrics for Next Fit

```

1   _____
2   CORE  1
3     core number      1
4     core load capacity 0.75
5     core rem capacity 0.25

```

```

6      Task in core      0
7      Task Utilization  0.5
8      _____
9      _____
10     CORE 1
11         core number      1
12         core load capacity 0.75
13         core rem capacity 0.0
14         Task in core      1
15         Task Utilization  0.25
16         _____
17         _____
18     CORE 2
19         core number      2
20         core load capacity 0.75
21         core rem capacity 0.25
22         Task in core      2
23         Task Utilization  0.5
24         _____
25         _____
26     CORE 2
27         core number      2
28         core load capacity 0.75
29         core rem capacity 0.04
30         Task in core      3
31         Task Utilization  0.2
32         _____
33
34     Number of processors used for NEXT FIT 2
35     Utilization factor of cores      [0.71, 0.75]
36     Maximum Utilization factor of cores 0.75
37     Minimum Utilization factor of cores 0.71

```

Listing 4.3: Metrics for First Fit

```

1      _____
2     CORE 1
3         core number      1
4         core load capacity 0.75
5         core rem capacity 0.25
6         Task in core      0
7         Task Utilization  0.5
8         _____
9         _____

```

```

10 CORE 1
11     core number          1
12     core load capacity  0.75
13     core rem capacity   0.0
14     Task in core        1
15     Task Utilization    0.25
16
17
18 CORE 2
19     core number          2
20     core load capacity  0.75
21     core rem capacity   0.25
22     Task in core        2
23     Task Utilization    0.5
24
25
26 CORE 2
27     core number          2
28     core load capacity  0.75
29     core rem capacity   0.04
30     Task in core        3
31     Task Utilization    0.2
32
33
34
35     Number of processors used for FIRST FIT 2
36     Utilization factor of cores    [0.71, 0.75]
37     Maximum Utilization factor of cores 0.75
38     Minimum Utilization factor of cores 0.71
39

```

Listing 4.4: Metrics for Best Fit

```

1
2 CORE 1
3     core number          1
4     core load capacity  0.75
5     core rem capacity   0.25
6     Task in core        0
7     Task Utilization    0.5
8
9
10 CORE 1
11     core number          1

```

```

12      core load capacity 0.75
13      core rem capacity 0.0
14      Task in core      1
15      Task Utilization  0.25
16      _____
17      _____
18  CORE 2
19      core number      2
20      core load capacity 0.75
21      core rem capacity 0.25
22      Task in core      2
23      Task Utilization  0.5
24      _____
25      _____
26  CORE 2
27      core number      2
28      core load capacity 0.75
29      core rem capacity 0.04
30      Task in core      3
31      Task Utilization  0.2
32      _____
33
34
35      Number of processors used for BEST FIT 2
36      Utilization factor of cores [0.71, 0.75]
37      Maximum Utilization factor of cores 0.75
38      Minimum Utilization factor of cores 0.71
39

```

4.2 Case 2 - Random task generation

	Period	WCET
Task 1	12	5
Task 2	15	7
Task 3	16	4
Task 4	16	4

Table 4.2: Test task 2

Listing 4.5: Data acquisition

```
1      number of tasks 4
2
3      TASK  1
4
5      Period      12
6      WCET        5
7      Utilization 0.41
8
9
10     TASK  2
11
12     Period      15
13     WCET        7
14     Utilization 0.46
15
16
17     TASK  3
18
19     Period      16
20     WCET        4
21     Utilization 0.25
22
23
24     TASK  4
25
26     Period      16
27     WCET        4
28     Utilization 0.25
29
30
```

```
sched_factor 0.75
```

Listing 4.6: Metrics for Next Fit

```
CORE 1
  core number          1
  core load capacity 0.75
  core rem capacity 0.34
  Task in core         0
  Task Utilization    0.41

CORE 2
  core number          2
  core load capacity 0.75
  core rem capacity 0.28
  Task in core         1
  Task Utilization    0.46

CORE 2
  core number          2
  core load capacity 0.75
  core rem capacity 0.03
  Task in core         2
  Task Utilization    0.25

CORE 3
  core number          3
  core load capacity 0.75
  core rem capacity 0.5
  Task in core         3
  Task Utilization    0.25

Number of processors used for NEXT FIT 3
Utilization factor of cores [0.25, 0.41, 0.72]
Maximum Utilization factor of cores 0.72
Minimum Utilization factor of cores 0.25
```


Listing 4.7: Metrics for First Fit

```

CORE 1
  core number      1
  core load capacity 0.75
  core rem capacity 0.34
  Task in core     0
  Task Utilization 0.41

```

```

CORE 2
  core number      2
  core load capacity 0.75
  core rem capacity 0.28
  Task in core     1
  Task Utilization 0.46

```

```

CORE 1
  core number      1
  core load capacity 0.75
  core rem capacity 0.09
  Task in core     2
  Task Utilization 0.25

```

```

CORE 2
  core number      2
  core load capacity 0.75
  core rem capacity 0.03
  Task in core     3
  Task Utilization 0.25

```

Number of processors used for FIRST FIT 2

Number of processors used for FIRST FIT 2

Utilization factor of cores [0.66, 0.71]

Maximum Utilization factor of cores 0.71

Minimum Utilization factor of cores 0.66

Listing 4.8: Metrics for Best Fit

```

1  _____
2  CORE 1
3      core number      1
4      core load capacity 0.75
5      core rem capacity 0.34
6      Task in core     0
7      Task Utilization 0.41
8  _____
9  _____
10 CORE 2
11     core number      2
12     core load capacity 0.75
13     core rem capacity 0.28
14     Task in core     1
15     Task Utilization 0.46
16  _____
17  _____
18 CORE 2
19     core number      2
20     core load capacity 0.75
21     core rem capacity 0.03
22     Task in core     3
23     Task Utilization 0.25
24  _____
25  _____
26 CORE 1
27     core number      1
28     core load capacity 0.75
29     core rem capacity 0.09
30     Task in core     2
31     Task Utilization 0.25
32  _____
33
34     Number of processors used for BEST FIT 2
35
36     Number of processors used for BEST FIT 2
37     Utilization factor of cores      [0.66, 0.71]
38     Maximum Utilization factor of cores 0.71
39     Minimum Utilization factor of cores 0.66
40

```



4.3 Case 3 - Random task generation

	Period	WCET
Task 1	9	5
Task 2	11	4
Task 3	16	7
Task 4	16	6

Table 4.3: Test task 3

Listing 4.9: Data acquisition

```
1  _____
2  TASK  1
3  _____
4  Period      9
5  WCET        5
6  Utilization 0.55
7  _____
8
9  TASK  2
10 _____
11 Period      11
12 WCET        4
13 Utilization 0.36
14 _____
15
16 TASK  3
17 _____
18 Period      16
19 WCET        7
20 Utilization 0.43
21 _____
22
23 TASK  4
24 _____
25 Period      16
26 WCET        6
27 Utilization 0.37
28 _____
29
```

Listing 4.10: Metrics for Next Fit Case 3

```

1  _____
2  CORE 1
3      core number      1
4      core load capacity 0.75
5      core rem capacity 0.19
6      Task in core      0
7      Task Utilization  0.55
8  _____
9  _____
10 CORE 2
11     core number      2
12     core load capacity 0.75
13     core rem capacity 0.39
14     Task in core      1
15     Task Utilization  0.36
16  _____
17  _____
18 CORE 3
19     core number      3
20     core load capacity 0.75
21     core rem capacity 0.32
22     Task in core      2
23     Task Utilization  0.43
24  _____
25  _____
26 CORE 4
27     core number      4
28     core load capacity 0.75
29     core rem capacity 0.38
30     Task in core      3
31     Task Utilization  0.37
32  _____
33
34     Number of processors used for NEXT FIT 4
35     Utilization factor of cores      [0.36, 0.37, 0.43, 0.56]
36     Maximum Utilization factor of cores 0.56
37     Minimum Utilization factor of cores 0.36
38

```

Listing 4.11: Metrics for First Fit case 3

```

1  _____
2  CORE 1

```

```

3      core number          1
4      core load capacity  0.75
5      core rem capacity   0.19
6      Task in core        0
7      Task Utilization    0.55
8      _____
9
10     CORE 2
11     core number          2
12     core load capacity   0.75
13     core rem capacity    0.39
14     Task in core         1
15     Task Utilization     0.36
16     _____
17
18     CORE 3
19     core number          3
20     core load capacity   0.75
21     core rem capacity    0.32
22     Task in core         2
23     Task Utilization     0.43
24     _____
25
26     CORE 2
27     core number          2
28     core load capacity   0.75
29     core rem capacity    0.02
30     Task in core         3
31     Task Utilization     0.37
32     _____
33
34     Number of processors used for FIRST FIT 3
35
36     Number of processors used for FIRST FIT 3
37     Utilization factor of cores [0.43, 0.56, 0.73]
38     Maximum Utilization factor of cores 0.73
39     Minimum Utilization factor of cores 0.43
40
41
42

```

Listing 4.12: Metrics for Best Fit case 3

```

2  CORE  1
3      core number          1
4      core load capacity  0.75
5      core rem capacity   0.19
6      Task in core        0
7      Task Utilization    0.55
8
9  _____
10
11 CORE  2
12     core number          2
13     core load capacity  0.75
14     core rem capacity   0.39
15     Task in core        1
16     Task Utilization    0.36
17
18 _____
19
20 CORE  3
21     core number          3
22     core load capacity  0.75
23     core rem capacity   0.32
24     Task in core        2
25     Task Utilization    0.43
26
27 _____
28
29 CORE  2
30     core number          2
31     core load capacity  0.75
32     core rem capacity   0.02
33     Task in core        3
34     Task Utilization    0.37
35
36 _____
37
38     Number of processors used for BEST FIT 3
39     Utilization factor of cores      [0.43, 0.56, 0.73]
40     Maximum Utilization factor of cores 0.73
41     Minimum Utilization factor of cores 0.43

```

Chapter 5

Conclusion

Three partitioning strategies for periodic tasks have been implemented successfully in **python3** as per the requirements. The program consists of the following parts.

- **Data acquisition:** The program takes input from the user as number of tasks, period and the worst case execution time of each task. The user data is stored in a python class data structure.
- **Random data set generation:** This is an optional part in which the program does not take any input from the user instead the task sets are randomly generated using the *random.randrange(lowerlimit, upperlimi)*.
- **Schedulability test:** The user data is tested for schedulability using the sufficient for rate monotonic scheduling.
- **Simulation:** This part partitioned the tasks based on next fit, first fit and best fit algorithm and the simulation results are displayed in the console. Certain observations were made after the simulation.
 - Next fit algorithm completes in $\mathcal{O}(n)$ time
 - First fit algorithm completes in $\mathcal{O}(n^2)$ time
 - Best fit algorithm completes in $\mathcal{O}(n \log n)$ time
 - $U_{max}^{BestFit} > U_{max}^{FirstFit} > U_{max}^{NextFit}$
 - $U_{min}^{NextFit} > U_{min}^{FirstFit} > U_{min}^{BestFit}$
- **Display of metrics:** The number of processors that used for each partitioning strategy is displayed. The minimum and maximum utilization factor of the cores are displayed. In addition to these metrics the details of each processor is displayed which contains the processor number, processor utilization, remaining utilization, task allocated to the current processor as well as the total available utilization of each processor is displayed in the console.

Chapter 6

Appendix

The partitioning program *partitioning.py* is developed in **python3**. To execute this program

```
1 $ git clone https://github.com/EnigmaRagesh/Multicore_Processor_Partitioning
2 $ cd RateMonotonic_Scheduler
3 $ python3 RM_scheduling.py
```

Listing 6.1: Display of Metrics

```
1 #!/usr/bin/env python3
2 # -----
3 # partitioning.py : Partitioning strategies:
4     # Best fist , First fit , Next fit
5 # Author: Ragesh RAMACHANDRAN
6 # -----
7 import json
8 import copy
9 import operator
10 import random
11 from sys import *
12 from math import gcd
13 import math
14 import numpy as np
15
16 # Data structure to store the task sets
17 class task:
18     def __init__(self,task_id=None, period=None, WCET=None, U=None):
19         self.task_id = task_id
20         self.period = period
21         self.WCET = WCET
22         self.U = U
```

```

23
24 # Data structure to store the processor data
25 class core:
26     def __init__(self, core_id=None, core_U=None, core_rem_U=None,
27                  task_ID=None, task_U=None):
28
29         self.core_id = core_id
30         self.core_U = core_U
31         self.core_rem_U = core_rem_U
32         self.task_ID = task_ID
33         self.task_U = task_U
34
35
36 def truncate(f, n):
37     return math.floor(f * 10 ** n) / 10 ** n
38
39 def hyperperiod():
40     temp = []
41     for i in range(n):
42         temp.append(tasks[i].period)
43     HP = temp[0]
44     for i in temp[1:]:
45         HP = HP*i//gcd(HP, i)
46     print ("\n\tHyperperiod:",HP)
47     return HP
48
49 def random_data():
50     global n      # Number of tasks to be partitioned
51     global tasks # List that stores the instances of tasks
52     tasks = []
53     random.seed() #Random seeding
54     n = random.randrange(2,7) #random in range of 2 and 7
55     print("number of tasks",n)
56
57     for i in range(n):
58         task_id = i
59         period = random.randrange(9,20) #random period
60         WCET = random.randrange(4,8)    #random WCET
61         u = WCET/period
62         # Limit U by 2 decimal places without rounding off
63         U = truncate(u,2)
64         tasks.append(task(task_id, period, WCET, U))
65

```

```

66 # Tasks are sorted based on their period and displayed
67 tasks = sorted(tasks, key=lambda tasks:tasks.period)
68 for i in range(n):
69     print("_____")
70     print("TASK   %d"%(i+1))
71     print("_____")
72     print("Period       ",tasks[i].period)
73     print("WCET         ",tasks[i].WCET)
74     print("Utilization",tasks[i].U)
75     print("_____")
76     print("\n\n")
77
78 def read_data():
79     global n      # Number of tasks to be partitioned
80     global tasks # List that stores the instances of tasks
81     tasks = []
82     n = int(input("\n \t\tEnter number of tasks:"))
83     # Read data from user
84     for i in range(n):
85         task_id = i
86         print("\nEnter Period of task T",i,":")
87         period = int(input())
88         print("Enter the WCET of task C",i,":")
89         WCET = int(input())
90         u = WCET/period
91         # Limit U by 2 decimal places without rounding off
92         U = truncate(u,2)
93         tasks.append(task(task_id,period,WCET,U))
94
95 # Tasks are sorted based on their period and displayed
96 tasks = sorted(tasks, key=lambda tasks:tasks.period)
97 for i in range(n):
98     print("_____")
99     print("TASK   %d"%(i+1))
100    print("_____")
101    print("Period       ",tasks[i].period)
102    print("WCET         ",tasks[i].WCET)
103    print("Utilization",tasks[i].U)
104    print("_____")
105    print("\n\n")
106
107 def schedulability():
108     global sched_factor

```

```

109 # Schedulability test for RM scheduling
110 sched_fac = n*(2**(1/n)-1)
111 # to limit only 2 decimal places without rounding off
112 sched_factor = truncate(sched_fac, 2)
113 print("\tsched_factor", sched_factor)
114
115 def NEXT_FIT():
116     # 'n' tasks and 'm' processors
117     m = 1 #core counter
118     c = 1 #initial core count
119     cores = [] #instance list of core class
120     core_rem = sched_factor #Store remaining U factor of core
121
122     for i in range(n):
123         # if a task cannot fit into same core
124         if(tasks[i].U > core_rem): #task does not fit into same core
125             c = c + 1 #task i is added to a new core
126             core_rem = sched_factor - tasks[i].U #calculate rem capacity
127             core_rem = truncate(core_rem,2)
128             cores.append(core(c, sched_factor, core_rem,i,tasks[i].U))
129             # If a task fit into a same core
130         else:
131             core_rem = core_rem - tasks[i].U #task run on same core
132             core_rem = truncate(core_rem,2)
133             cores.append(core(c, sched_factor, core_rem,i,tasks[i].U))
134     # if there are no new cores added then default value = 1
135     if(c>m):
136         m = c
137     # sorting the processors based on core_id
138     cores = sorted(cores, key=lambda cores:cores.core_id )
139     for i in range(len(cores)):
140         print("_____")
141         print("\nCORE   %d"%cores[i].core_id)
142         print("\tcore number       ",cores[i].core_id)
143         print("\tcore load capacity",cores[i].core_U)
144         print("\tcore rem capacity ",cores[i].core_rem_U)
145         print("\tTask in core       ",cores[i].task_ID)
146         print("\tTask Utilization   ",cores[i].task_U)
147         print("_____")
148     print("\n\tNumber of processors used for NEXT FIT",m)
149     display_metrics(cores)
150
151 def FIRST_FIT():

```

```

152 # 'n' tasks and 'res' processors
153 global need
154 res = 0
155 core_remain = [0]*n# array to store remaining space in cores
156 cores = [] #instance list of core class
157 core_rem = sched_factor #Store remaining U factor of core
158
159 for i in range(n):
160     need = 0
161     # Find the first core that can schedule the task
162     for j in range(res):
163         if core_remain[j] >= tasks[i].U:
164             core_remain[j] = tasks[i].U - core_remain[j]
165             cores.append(core(res,
166                             sched_factor,
167                             truncate(tasks[i].U - core_remain[j], 2),
168                             i,
169                             tasks[i].U))
170             break
171     else:
172         need = need + 1
173
174     # If no core can schedule the task add new core
175     if need == res:
176         core_remain[res] = sched_factor - tasks[i].U
177         res = res + 1 #task i is added to a new core
178         cores.append(core(res,
179                             sched_factor,
180                             truncate(sched_factor - tasks[i].U, 2),
181                             i,
182                             tasks[i].U))
183
184 for i in range(len(cores)):
185     print("_____")
186     print("\nCORE   %d"%cores[i].core_id)
187     print("\tcore number       ",cores[i].core_id)
188     print("\tcore load capacity",cores[i].core_U)
189     print("\tcore rem capacity ",cores[i].core_rem_U)
190     print("\tTask in core       ",cores[i].task_ID)
191     print("\tTask Utilization   ",cores[i].task_U)
192     print("_____")
193 print("\n\tNumber of processors used for FIRST FIT",res)
194 dispaly_metrics(cores)

```

```

195
196 def BEST_FIT():
197     # 'n' tasks and 'm' processors
198     res = 0
199     core_remain = [0]*n # array to store remaining space in cores
200     cores = [] #instance list of core class
201     core_rem = sched_factor #Store remaining U factor of core
202     count = sched_factor
203
204     for i in range(n):
205         min = count + 1
206         for j in range(res):
207             if (core_remain[j] >= tasks[i].U
208                 and core_remain[j] - tasks[i].U < min):
209                 min = core_remain[j] - tasks[i].U
210                 c = j
211                 break
212
213         if min == count + 1:
214             core_remain[res] = sched_factor - tasks[i].U
215             res = res + 1
216             cores.append(core(res,
217                             sched_factor,
218                             truncate(sched_factor - tasks[i].U, 2),
219                             i,
220                             tasks[i].U))
221         else:
222             core_remain[c] = tasks[i].U - core_remain[c]
223             cores.append(core(res,
224                             sched_factor,
225                             truncate(tasks[i].U - core_remain[c], 2),
226                             i,
227                             tasks[i].U))
228
229     for i in range(len(cores)):
230         print("_____")
231         print("\nCORE   %d"%cores[i].core_id)
232         print("\tcore number       ",cores[i].core_id)
233         print("\tcore load capacity",cores[i].core_U)
234         print("\tcore rem capacity ",cores[i].core_rem_U)
235         print("\tTask in core       ",cores[i].task_ID)
236         print("\tTask Utilization   ",cores[i].task_U)
237         print("_____")

```

```

238 print("\n\tNumber of processors used for BEST FIT",res)
239 dispaly_metrics(cores)
240
241 def dispaly_metrics(cores):
242     core_buff = [] #Store the utilization of cores
243     merged_cores = []#Store the non-duplicate cores
244     id_list = []#Stores id list of cores
245     id_merged = []#Stores id list of merged cores
246
247     # The cores are sorted based on the core_id
248     cores = sorted(cores, key=lambda cores:cores.core_id )
249     for i in range(len(cores)):
250         id_list.append(cores[i].core_id)
251     # Metrics are calculated here
252
253     # Inorder to remove the multiple instances of core
254     # with same core_id they are merged
255     id_merged = [i for i, x in enumerate(id_list)
256                  if i == len(id_list) - 1 or x != id_list[i + 1]]
257     # A merged_core array is created to store the core_id
258     for i in id_merged:
259         merged_cores.append(cores[i])
260
261     # Display of main metrics
262     for i in range(len(merged_cores)): #truncate the utilization value
263         core_buff.append(truncate(merged_cores[i].core_U
264                                   - merged_cores[i].core_rem_U,2))
265
266     core_buff.sort() #sorting the U list for finding max and min values
267     print("Utilization factor of cores      ", core_buff)
268     print("Maximum Utilization factor of cores", core_buff[-1])
269     print("Minimum Utilization factor of cores", core_buff[0])
270
271 if __name__ == '__main__':
272     random_data()#Random task set generation
273     read_data() #Reads taskset from user
274     hp = hyperperiod()#Calculates hyperperiod
275     schedulability()#Check for feasibility
276     NEXT_FIT()#Next fit partitioning
277     FIRST_FIT()#First fit partitioning
278     BEST_FIT()#Best fit partitioning

```