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

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,..,n} and each tasks have 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

procedure Greedy algorithm
  Input:$\tau_1, \tau_2 \dots \tau_n$
  Sort the task sets $\tau_1, \tau_2 \dots \tau_n$ according to period, priority or utilization
  j = 0
  for *i=1* to *n* **do**
    Determine the utilization of each task
    if Task *i* fits into processor *j* **then**
      Fit i into j
    else
      j = j +1
      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

```
struct task
{
    int task_id
    int period
    int WCET
    float U
}
```

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

```
struct core
{
    int core_id
    float core_U
    float core_rem_U
}
```

- 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.

```
main()
{
   read_data()
   random_data()
   hp = hyperperiod()
   schedulability()
   NEXT_FIT()
   FIRST_FIT()
   BEST_FIT()
}
```

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

```
class task:
  def __init__(self,task_id = None,period = None,WCET = None,U = None):
    self.task_id = task_id
    self.period  = period
    self.WCET    = WCET
    self.U       = U

```

Listing 3.2: Data acquisition

```

def read_data():
  global n
  global tasks
  tasks = []
  n = int(input("\n \t\tEnter number of tasks:"))
  # Read data from user
  for i in range(n):
    task_id = i
    print("\nEnter Period of task T",i,":")
    period = int(input())
    print("Enter the WCET of task C",i,":")
    WCET = int(input())
    u =  WCET/period
    #Truncating the float result to 2 decimal places
    U = truncate(u,2)
    tasks.append(task(task_id,period,WCET,U))

  # Tasks are sorted based on their period
  tasks = sorted(tasks, key=lambda tasks:tasks.period)
  for i in range(n):
    print("-------------------------------")
    print("TASK  %d"%(i+1))
    print("-------------------------------")
    print("Period      ",tasks[i].period)
    print("WCET        ",tasks[i].WCET)
    print("Utilization",tasks[i].U)
    print("-------------------------------")
    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

```python
def random_data():
  global n     # Number of tasks to be partitioned
  global tasks # List that stores the instances of tasks
  tasks = []
  random.seed()
  n = random.randrange(2,7)
  print("number of tasks",n)

  for i in range(n):
    task_id = i
    period = random.randrange(9,20)
    WCET = random.randrange(4,8)
    u =  WCET/period
    # Limit U by 2 decimal places without rounding off
    U = truncate(u,2)
    tasks.append(task(task_id,period,WCET,U))

  # Tasks are sorted based on their period and displayed
  tasks = sorted(tasks, key=lambda tasks:tasks.period)
  for i in range(n):
    print("------------------------------")
    print("TASK  %d"%(i+1))
    print("------------------------------")
    print("Period      ",tasks[i].period)
    print("WCET        ",tasks[i].WCET)
    print("Utilization",tasks[i].U)
    print("------------------------------")
    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.

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

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

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

Listing 3.4: Schedulability test

```
def schedulability():
  global sched_factor
  # Schedulability test for RM scheduling
  sched_fac = n*(2**(1/n)-1)
  # to limit only 2 decimal places without rounding off
  sched_factor = truncate(sched_fac, 2)
  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

```
class core:
  def __init__(self,core_id=0,core_U=0,core_rem_U=0,task_ID=0,task_U=0):
    self.core_id    = core_id
    self.core_U     = core_U
    self.core_rem_U = core_rem_U
    self.task_ID    = task_ID
    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

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

```
        core_rem = core_rem - tasks[i].U #task run on same core
        core_rem = truncate(core_rem,2)
        cores.append(core(c, sched_factor, core_rem,i,tasks[i].U))
    # if there are no new cores added then default value = 1
    if(c>m):
        m = c
    # sorting the processors based on core_id
    cores = sorted(cores, key=lambda cores:cores.core_id )
    for i in range(len(cores)):
        id_list.append(cores[i].core_id)
        print("------------------------------")
        print("\nCORE  %d"%cores[i].core_id)
        print("\tcore number        ",cores[i].core_id)
        print("\tcore load capacity",cores[i].core_U)
        print("\tcore rem capacity ",cores[i].core_rem_U)
        print("\tTask in core       ",cores[i].task_ID)
        print("\tTask Utilization   ",cores[i].task_U)
        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

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

```
core_buff = []          #Store the utilization of cores
merged_cores = []       #Store the non-duplicate cores
id_list = []      #Stores id list of cores
id_merged = []      #Stores id list of merged cores

for i in range(n):
  need = 0
  # Find the first core that can schedule the task
  for j in range(res):
    if core_remain[j] >= tasks[i].U:
      core_remain[j] = tasks[i].U - core_remain[j]
          cores.append(core(res,
          sched_factor,
          truncate(tasks[i].U - core_remain[j] ,2),
          i,
          tasks[i].U))
      break
    else:
      need = need +1

  # If no core can schedule the task add new core
  if need == res:
    core_remain[res] = sched_factor - tasks[i].U
    res = res +1   #task i is added to a new core
    cores.append(core(res,
            sched_factor,
            truncate(sched_factor - tasks[i].U,2),
            i,
            tasks[i].U))

for i in range(len(cores)):
  print("-----------------------------------")
  print("\nCORE  %d"%cores[i].core_id)
  print("\tcore number       ",cores[i].core_id)
  print("\tcore load capacity",cores[i].core_U)
  print("\tcore rem capacity ",cores[i].core_rem_U)
  print("\tTask in core      ",cores[i].task_ID)
  print("\tTask Utilization  ",cores[i].task_U)
  print("-----------------------------------")
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

```
def BEST_FIT():
  # 'n' tasks and 'm' processors
  res = 0
  core_remain = [0]*n # array to store remaining space in cores
  cores = []        #instance list of core class
  core_rem = sched_factor #Store remaining U factor of core
  core_buff = []          #Store the utilization of cores
  merged_cores = []       #Store the non-duplicate cores
  id_list = []      #Stores id list of cores
  id_merged = []      #Stores id list of merged cores
  count = sched_factor

  for i in range(n):
    min = count +1
    for j in range(res):
      if core_remain[j] >= tasks[i].U and core_remain[j]-tasks[i].U < min:
        min = core_remain[j] - tasks[i].U
        c = j
        break

    if min == count+1:
      core_remain[res] = sched_factor - tasks[i].U
      res = res  + 1
      cores.append(core(res,
                        sched_factor,
                        truncate(sched_factor - tasks[i].U,2),
```

```
                        i,
                        tasks[i].U))
    else:
      core_remain[c] = tasks[i].U - core_remain[c]
      cores.append(core(res,
                       sched_factor,
                       truncate(tasks[i].U - core_remain[c],2),
                       i,
                       tasks[i].U))

  for i in range(len(cores)):
    print("--------------------------------")
    print("\nCORE  %d"%cores[i].core_id)
    print("\tcore number       ",cores[i].core_id)
    print("\tcore load capacity",cores[i].core_U)
    print("\tcore rem capacity ",cores[i].core_rem_U)
    print("\tTask in core      ",cores[i].task_ID)
    print("\tTask Utilization  ",cores[i].task_U)
    print("--------------------------------")
  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

```
# sorting the processors based on core_id
  cores = sorted(cores, key=lambda cores:cores.core_id )
  for i in range(len(cores)):
    id_list.append(cores[i].core_id)

  # Metrics are calculated here
  # Inorder to remove the multiple instances of
  # core with same core_id they are merged
  id_merged = [i for i, x in enumerate(id_list)
        if i == len(id_list) - 1 or x != id_list[i + 1]]
  for i in id_merged:
    merged_cores.append(cores[i])

  # Display of main metrics
```

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

```
    Enter number of tasks:4

Enter Period of task T 0 :
8
Enter the WCET of task C 0 :
4

Enter Period of task T 1 :
8
Enter the WCET of task C 1 :
2

Enter Period of task T 2 :
8
Enter the WCET of task C 2 :
4
```

```
Enter Period of task T 3 :
10
Enter the WCET of task C 3 :
2

--------------------
TASK 1
--------------------
Period      8
WCET        4
Utilization 0.5
--------------------
--------------------
TASK 2
--------------------
Period      8
WCET        2
Utilization 0.25
--------------------
--------------------
TASK 3
--------------------
Period      8
WCET        4
Utilization 0.5
--------------------
--------------------
TASK 4
--------------------
Period      10
WCET        2
Utilization 0.2
--------------------


```

Listing 4.2: Metrics for Next Fit

```
--------------------------------------
CORE 1
  core number        1
  core load capacity 0.75
  core rem capacity  0.25
```

```
    Task in core        0
    Task Utilization    0.5
  ------------------------------------
  ------------------------------------
  CORE  1
    core number         1
    core load capacity  0.75
    core rem capacity   0.0
    Task in core        1
    Task Utilization    0.25
  ------------------------------------
  ------------------------------------
  CORE  2
    core number         2
    core load capacity  0.75
    core rem capacity   0.25
    Task in core        2
    Task Utilization    0.5
  ------------------------------------
  ------------------------------------
  CORE  2
    core number         2
    core load capacity  0.75
    core rem capacity   0.04
    Task in core        3
    Task Utilization    0.2
  ------------------------------------

    Number of processors used for NEXT FIT 2
  Utilization factor of cores      [0.71, 0.75]
  Maximum Utilization factor of cores 0.75
  Minimum Utilization factor of cores 0.71
```

Listing 4.3: Metrics for First Fit

```
  ------------------------------------
  CORE  1
      core number         1
      core load capacity  0.75
      core rem capacity   0.25
      Task in core        0
      Task Utilization    0.5
  ------------------------------------
  ------------------------------------
```

```
CORE  1
    core number        1
    core load capacity 0.75
    core rem capacity  0.0
    Task in core       1
    Task Utilization   0.25
________________________________
________________________________
CORE  2
    core number        2
    core load capacity 0.75
    core rem capacity  0.25
    Task in core       2
    Task Utilization   0.5
________________________________
________________________________
CORE  2
    core number        2
    core load capacity 0.75
    core rem capacity  0.04
    Task in core       3
    Task Utilization   0.2
________________________________


     Number of processors used for FIRST FIT 2
 Utilization factor of cores      [0.71, 0.75]
Maximum Utilization factor of cores 0.75
Minimum Utilization factor of cores 0.71

```

Listing 4.4: Metrics for Best Fit

```
________________________________
CORE  1
    core number        1
    core load capacity 0.75
    core rem capacity  0.25
    Task in core       0
    Task Utilization   0.5
________________________________
________________________________
CORE  1
    core number        1
```

```
    core load capacity 0.75
    core rem capacity  0.0
    Task in core       1
    Task Utilization   0.25
____________________________________
____________________________________
CORE  2
    core number        2
    core load capacity 0.75
    core rem capacity  0.25
    Task in core       2
    Task Utilization   0.5
____________________________________
____________________________________
CORE  2
    core number        2
    core load capacity 0.75
    core rem capacity  0.04
    Task in core       3
    Task Utilization   0.2
____________________________________


     Number of processors used for BEST FIT 2
 Utilization factor of cores      [0.71, 0.75]
Maximum Utilization factor of cores 0.75
Minimum Utilization factor of cores 0.71

```

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

```
    number of tasks 4
----------------------
TASK  1
----------------------
Period       12
WCET         5
Utilization  0.41
----------------------
----------------------
TASK  2
----------------------
Period       15
WCET         7
Utilization  0.46
----------------------
----------------------
TASK  3
----------------------
Period       16
WCET         4
Utilization  0.25
----------------------
----------------------
TASK  4
----------------------
Period       16
WCET         4
Utilization  0.25
----------------------
```

```
    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

```
———————————————————————————————
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       3
  Task Utilization   0.25
———————————————————————————————
———————————————————————————————
CORE  1
  core number        1
  core load capacity 0.75
  core rem capacity  0.09
  Task in core       2
  Task Utilization   0.25
———————————————————————————————

  Number of processors used for BEST FIT 2

  Number of processors used for BEST FIT 2
 Utilization factor of cores      [0.66, 0.71]
Maximum Utilization factor of cores 0.71
Minimum Utilization factor of cores 0.66

```

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

```
--------------------
TASK  1
--------------------
Period        9
WCET          5
Utilization   0.55
--------------------
--------------------
TASK  2
--------------------
Period        11
WCET          4
Utilization   0.36
--------------------
--------------------
TASK  3
--------------------
Period        16
WCET          7
Utilization   0.43
--------------------
--------------------
TASK  4
--------------------
Period        16
WCET          6
Utilization   0.37
--------------------

```

Listing 4.10: Metrics for Next Fit Case 3

```
————————————————————————————————
CORE  1
  core number        1
  core load capacity 0.75
  core rem capacity  0.19
  Task in core       0
  Task Utilization   0.55
————————————————————————————————
————————————————————————————————
CORE  2
  core number        2
  core load capacity 0.75
  core rem capacity  0.39
  Task in core       1
  Task Utilization   0.36
————————————————————————————————
————————————————————————————————
CORE  3
  core number        3
  core load capacity 0.75
  core rem capacity  0.32
  Task in core       2
  Task Utilization   0.43
————————————————————————————————
————————————————————————————————
CORE  4
  core number        4
  core load capacity 0.75
  core rem capacity  0.38
  Task in core       3
  Task Utilization   0.37
————————————————————————————————

  Number of processors used for NEXT FIT 4
 Utilization factor of cores      [0.36, 0.37, 0.43, 0.56]
Maximum Utilization factor of cores 0.56
Minimum Utilization factor of cores 0.36

```

Listing 4.11: Metrics for First Fit case 3

```
————————————————————————————————
CORE  1
```

```
    core number        1
    core load capacity 0.75
    core rem capacity  0.19
    Task in core       0
    Task Utilization   0.55
-----------------------------------
-----------------------------------
CORE  2
    core number        2
    core load capacity 0.75
    core rem capacity  0.39
    Task in core       1
    Task Utilization   0.36
-----------------------------------
-----------------------------------
CORE  3
    core number        3
    core load capacity 0.75
    core rem capacity  0.32
    Task in core       2
    Task Utilization   0.43
-----------------------------------
-----------------------------------
CORE  2
    core number        2
    core load capacity 0.75
    core rem capacity  0.02
    Task in core       3
    Task Utilization   0.37
-----------------------------------

   Number of processors used for FIRST FIT 3

   Number of processors used for FIRST FIT 3
 Utilization factor of cores      [0.43, 0.56, 0.73]
Maximum Utilization factor of cores 0.73
Minimum Utilization factor of cores 0.43
```

Listing 4.12: Metrics for Best Fit case 3

```
-----------------------------------
```

```
CORE  1
  core number        1
  core load capacity 0.75
  core rem capacity  0.19
  Task in core       0
  Task Utilization   0.55
________________________________________
________________________________________
CORE  2
  core number        2
  core load capacity 0.75
  core rem capacity  0.39
  Task in core       1
  Task Utilization   0.36
________________________________________
________________________________________
CORE  3
  core number        3
  core load capacity 0.75
  core rem capacity  0.32
  Task in core       2
  Task Utilization   0.43
________________________________________
________________________________________
CORE  2
  core number        2
  core load capacity 0.75
  core rem capacity  0.02
  Task in core       3
  Task Utilization   0.37
________________________________________

  Number of processors used for BEST FIT 3
 Utilization factor of cores      [0.43, 0.56, 0.73]
Maximum Utilization factor of cores 0.73
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 $\mathcal{O}(nlogn)$ 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

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

Listing 6.1: Display of Metrics

```
#!/usr/bin/env python3
# ———————————————————————————————
# partitioning.py : Partitioning strategies:
      # Best fist , First fit , Next fit
# Author: Ragesh RAMACHANDRAN
# ———————————————————————————————
import json
import copy
import operator
import random
from sys import *
from math import gcd
import math
import numpy as np

# Data structure to store the task sets
class task:
  def __init__(self,task_id=None, period=None, WCET=None, U=None):
    self.task_id = task_id
    self.period = period
    self.WCET = WCET
    self.U = U
```

```
# Data structure to store the processor data
class core:
  def __init__(self,core_id=None,core_U=None,core_rem_U=None,
                                    task_ID=None,task_U=None):

    self.core_id = core_id
    self.core_U = core_U
    self.core_rem_U = core_rem_U
    self.task_ID =  task_ID
    self.task_U = task_U


def truncate(f, n):
  return math.floor(f * 10 ** n) / 10 ** n

def hyperperiod():
  temp = []
  for i in range(n):
    temp.append(tasks[i].period)
  HP = temp[0]
  for i in temp[1:]:
    HP = HP*i//gcd(HP, i)
  print ("\n\tHyperperiod:",HP)
  return HP

def random_data():
  global n     # Number of tasks to be partitioned
  global tasks # List that stores the instances of tasks
  tasks = []
  random.seed() #Random seeding
  n = random.randrange(2,7)  #random in range of 2 and 7
  print("number of tasks",n)

  for i in range(n):
    task_id = i
    period = random.randrange(9,20) #random period
    WCET = random.randrange(4,8)    #random WCET
    u =  WCET/period
    # Limit U by 2 decimal places without rounding off
    U = truncate(u,2)
    tasks.append(task(task_id,period,WCET,U))

```

```
  # Tasks are sorted based on their period and displayed
  tasks = sorted(tasks, key=lambda tasks:tasks.period)
  for i in range(n):
    print("------------------------------")
    print("TASK  %d"%(i+1))
    print("------------------------------")
    print("Period      ",tasks[i].period)
    print("WCET        ",tasks[i].WCET)
    print("Utilization",tasks[i].U)
    print("------------------------------")
    print("\n\n")

def read_data():
  global n     # Number of tasks to be partitioned
  global tasks # List that stores the instances of tasks
  tasks = []
  n = int(input("\n \t\tEnter number of tasks:"))
  # Read data from user
  for i in range(n):
    task_id = i
    print("\nEnter Period of task T",i,":")
    period = int(input())
    print("Enter the WCET of task C",i,":")
    WCET = int(input())
    u =  WCET/period
    # Limit U by 2 decimal places without rounding off
    U = truncate(u,2)
    tasks.append(task(task_id,period,WCET,U))

  # Tasks are sorted based on their period and displayed
  tasks = sorted(tasks, key=lambda tasks:tasks.period)
  for i in range(n):
    print("------------------------------")
    print("TASK  %d"%(i+1))
    print("------------------------------")
    print("Period      ",tasks[i].period)
    print("WCET        ",tasks[i].WCET)
    print("Utilization",tasks[i].U)
    print("------------------------------")
    print("\n\n")

def schedulability():
  global sched_factor
```

```python
  # Schedulability test for RM scheduling
  sched_fac = n*(2**(1/n)-1)
  # to limit only 2 decimal places without rounding off
  sched_factor = truncate(sched_fac, 2)
  print("\tsched_factor", sched_factor)

def NEXT_FIT():
  # 'n' tasks and 'm' processors
  m = 1 #core counter
  c = 1 #initial core count
  cores = []#instance list of core class
  core_rem = sched_factor #Store remaining U factor of core

  for i in range(n):
    # if a task cannot fit into same core
    if(tasks[i].U > core_rem): #task does not fit into same core
      c = c + 1  #task i is added to a new core
      core_rem = sched_factor - tasks[i].U  #calculate rem capacity
      core_rem = truncate(core_rem,2)
      cores.append(core(c, sched_factor, core_rem,i,tasks[i].U))
     # If a task fit into a same core
    else:
      core_rem = core_rem - tasks[i].U #task run on same core
      core_rem = truncate(core_rem,2)
      cores.append(core(c, sched_factor, core_rem,i,tasks[i].U))
  # if there are no new cores added then default value = 1
  if(c>m):
     m = c
  # sorting the processors based on core_id
  cores = sorted(cores, key=lambda cores:cores.core_id )
  for i in range(len(cores)):
    print("-----------------------------------")
    print("\nCORE  %d"%cores[i].core_id)
    print("\tcore number       ",cores[i].core_id)
    print("\tcore load capacity",cores[i].core_U)
    print("\tcore rem capacity ",cores[i].core_rem_U)
    print("\tTask in core      ",cores[i].task_ID)
    print("\tTask Utilization  ",cores[i].task_U)
    print("-----------------------------------")
  print("\n\tNumber of processors used for NEXT FIT",m)
  dispaly_metrics(cores)

def FIRST_FIT():
```

```python
# 'n' tasks and 'res' processors
global need
res = 0
core_remain = [0]*n# array to store remaining space in cores
cores = [] #instance list of core class
core_rem = sched_factor #Store remaining U factor of core

for i in range(n):
  need = 0
  # Find the first core that can schedule the task
  for j in range(res):
    if core_remain[j] >= tasks[i].U:
      core_remain[j] = tasks[i].U - core_remain[j]
      cores.append(core(res,
      sched_factor,
      truncate(tasks[i].U - core_remain[j] ,2),
      i,
      tasks[i].U))
      break
    else:
      need = need +1

  # If no core can schedule the task add new core
  if need == res:
    core_remain[res] = sched_factor - tasks[i].U
    res = res +1 #task i is added to a new core
    cores.append(core(res,
      sched_factor,
      truncate(sched_factor - tasks[i].U,2),
      i,
      tasks[i].U))

for i in range(len(cores)):
  print("----------------------------------------")
  print("\nCORE  %d"%cores[i].core_id)
  print("\tcore number       ",cores[i].core_id)
  print("\tcore load capacity",cores[i].core_U)
  print("\tcore rem capacity ",cores[i].core_rem_U)
  print("\tTask in core      ",cores[i].task_ID)
  print("\tTask Utilization  ",cores[i].task_U)
  print("----------------------------------------")
print("\n\tNumber of processors used for FIRST FIT",res)
dispaly_metrics(cores)
```

```

def BEST_FIT():
  # 'n' tasks and 'm' processors
  res = 0
  core_remain = [0]*n # array to store remaining space in cores
  cores = []#instance list of core class
  core_rem = sched_factor #Store remaining U factor of core
  count = sched_factor

  for i in range(n):
    min = count +1
    for j in range(res):
      if (core_remain[j] >= tasks[i].U
        and core_remain[j]-tasks[i].U < min):
        min = core_remain[j] - tasks[i].U
        c = j
        break

    if min == count+1:
      core_remain[res] = sched_factor - tasks[i].U
      res = res  + 1
      cores.append(core(res,
        sched_factor,
        truncate(sched_factor - tasks[i].U,2),
        i,
        tasks[i].U))
    else:
      core_remain[c] = tasks[i].U - core_remain[c]
      cores.append(core(res,
        sched_factor,
        truncate(tasks[i].U - core_remain[c],2),
        i,
        tasks[i].U))

  for i in range(len(cores)):
    print("----------------------------------------")
    print("\nCORE  %d"%cores[i].core_id)
    print("\tcore number       ",cores[i].core_id)
    print("\tcore load capacity",cores[i].core_U)
    print("\tcore rem capacity ",cores[i].core_rem_U)
    print("\tTask in core      ",cores[i].task_ID)
    print("\tTask Utilization  ",cores[i].task_U)
    print("----------------------------------------")
```

```
  print("\n\tNumber of processors used for BEST FIT",res)
  dispaly_metrics(cores)

def dispaly_metrics(cores):
  core_buff = [] #Store the utilization of cores
  merged_cores = []#Store the non-duplicate cores
  id_list = []#Stores id list of cores
  id_merged = []#Stores id list of merged cores

  # The cores are sorted based on the core_id
  cores = sorted(cores, key=lambda cores:cores.core_id )
  for i in range(len(cores)):
    id_list.append(cores[i].core_id)
  # Metrics are calculated here

  # Inorder to remove the multiple instances of core
  #  with same core_id they are merged
  id_merged = [i for i, x in enumerate(id_list)
        if i == len(id_list) - 1 or x != id_list[i + 1]]
  # A merged_core array is created to store the core_id
  for i in id_merged:
    merged_cores.append(cores[i])

  # Display of main metrics
  for i in range(len(merged_cores)): #truncate the utilization value
    core_buff.append(truncate(merged_cores[i].core_U
              - merged_cores[i].core_rem_U,2))

  core_buff.sort() #sorting the U list for finding max and min values
  print("Utilization factor of cores      ", core_buff)
  print("Maximum Utilization factor of cores", core_buff[-1])
  print("Minimum Utilization factor of cores", core_buff[0])

if __name__ == '__main__':
  random_data()#Random task set generation
  read_data() #Reads taskset from user
  hp = hyperperiod()#Calculates hyperperiod
  schedulability()#Check for feasibility
  NEXT_FIT()#Next fit partitioning
  FIRST_FIT()#First fit partitioning
  BEST_FIT()#Best fit partitioning
```