PROJECT REPORT

done under

the guidance of

Dr Biju K Ravindran

Submitted by

Vaibhav Ranjith

(2017A8PS0556G)

In partial fulfilment for the award of degree in

Bachelor of Engineering:

Electronics and Instrumentation

Course: *Design Project*

Course Code: **INSTR F377**

Designing of cache memory to improve WCET in real-time RISC-V embedded system.

BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI

December,2019

Abstract

This report discusses the various principles of Real-time Operating Systems and how the operating system schedule a set of tasks. We go through the concepts of RTOS and dive into the popular scheduling algorithms.

ACKNOWLEDGEMENT

I feel deeply honoured to express my sincere thanks to Dr Biju K Ravindran, Professor in the computer science department at BITS Pilani, KK Birla Goa campus, for providing me an opportunity to undertake this project. His sincere guidance gave me deep insights into the field of system design making me more enthusiastic for learning more.

I would Like to express my gratitude to the members of the BITS Pilani, Goa Campus for their kind co-operation and encouragement which helped me in the partial completion of my project.

Last but not the least I would like to thank my family, without their constant emotional support I cannot imagine succeeding in any of my endeavours.

CONTENTS

CHAPTER 1

REAL TIME OPERATING SYSTEMS

Before defining a real time system. We have to talk about two terms, they are namely jobs and tasks. In the context of real time systems they are not the same. A job is a unit of work that is scheduled to be executed in the processor. Many units of work or a collection of related jobs provides a system function, a task. For example calculating the FFT of a signal, comparing a temperature to a measured temperature in an air conditioner or signalling the interlocks in an industry to execute because of a measured value by a sensor in the field going over a threshold value are all jobs. The majority of this report will talk about how to schedule jobs on a uni-core processor.

So basically task is a static entity and job is a dynamic entity. Analogous to and object and an instant of the object. Where object is to task and job is to instant.

1.1Hard real time systems

The "strictness" of the deadline of jobs running in the processor determines if a given system is a hard real time system or not. A job is first released for execution, then it waits for execution in a data structure, most likely a queue. It enters the processor for execution according to a set of decisions. So a deadline imposed on the job is the time relative to the time of release at which the job is expected to finish execution.

A deadline is a hard deadline if the results obtained after missing the deadline is of no use. Missing deadlines may be fatal in some cases and may have strong consequences. We give a probabilistic measure of how hard a deadline is, like a job completes it's job 99.99% of the time.

The execution time of a job may not be known during releasing but we might be able to find the lower and upper bound on execution time. To explain scheduling in this report we assume that we know these bounds on execution time.

1.2 Periodic task model

Periodic task model is a well-known deterministic model. We will be using this model to understand the scheduling algorithm. In this model every job that happens in regular interval of time is modelled as a *periodic tasks.* Most scheduling algorithms with this model hold to be true if the inter release time between two jobs at least have a lower bound at the period of the task. We will call the tasks in a system as $T_{1,}$ $T_{2,}$ T_3...T_n. The individual jobs in a task is referred to as $J_{i,1}$, $J_{i,2}$, and so $J_{i,k}$ being the kth job in the task.

Sometimes a real-time system is may require to respond to external events, and to respond it executes aperiodic or sporadic jobs whose release times are not known a priori. In the periodic task model these responses are modelled as *aperiodic* and *sporadic* tasks. An aperiodic tasks are tasks whose jobs, may come in highly irregular intervals of time, whose deadline is not as hard as the deadline imposed upon periodic tasks. Sporadic jobs in the other hand have hard deadline, but their are alike to aperiodic tasks in their job distribution.

Data dependency may constraint scheduling of jobs in any order. Such jobs are said to have a *precedence constraint.* Otherwise if the jobs can execute in any order we say they are *independent*. In this report we don't consider precedence constraint and assume that all the jobs released by tasks are independent.

So now that the essential terms and constraints are discussed. We can discuss the problem of scheduling being addressed in adjacent chapters. We can imagine a system to be a mathematical set of tasks where each task is a set of jobs. Each job in these tasks are either periodic, aperiodic or sporadic jobs. So scheduling is a mathematical problem of how to validate and schedule these jobs in a certain priority to ensure smooth operation of the system.

Chapter 2 will focus on the rate monotonic algorithm(RMA) of scheduling periodic tasks. We will also see how a scheduler was implemented in C programming language of a set of tasks. Chapter 3 will discuss Earliest deadline first scheduling of periodic tasks. Chapter 4 discusses the scheduling of aperiodic tasks. We will also look into a scheduler that I implemented to schedule aperiodic jobs with periodic jobs using deferred servers a bandwidth preserving server with RMA scheduling.

CHAPTER 2
RATE MONOTONIC ALGORITHM

This chapter discusses the priority driven rate monotonic algorithm RMA to schedule periodic tasks on a processor before which we take the assumptions that the tasks are independent and there are no periodic and aperiodic tasks. We also assume that the jobs are ready for execution as soon as they are released, they are premptable and never suspends itself.

When a tasks releases a job and ask the scheduler to schedule it they provide the parameters of period, execution time and relative deadline. Priorities of jobs within the ready queue are calculated using these parameters.

Dynamic and *Static* systems: A multiprocessor priority driven system is either dynamic or static. In static systems all the tasks are partitioned into subsystems and each subsystems are assigned to a processor. While in a dynamic system jobs ready for execution are scheduled in a common priority queue and dispatched when a processor becomes free. For fluctuating traffic dynamic priority looks as the more efficient solution but testing dynamic systems are more time consuming hence most hard real-time systems are of static priority. We reduce this problem to a uniprocessor case.

Priority driven algorithms can be classified as fixed priority or dynamic priority scheduling. In fixed-priority algorithm each job in task are given the same priority whereas in a dynamic priority algorithm each job in a task are given different priorities. In a fixed priority system once a priority is assigned to a job upon release it’s priority doesn’t change relative to the other jobs in the ready queue. Rate monotonic algorithm is a fixed-priority algorithm.

Rate monotonic algorithm assigns priorities to the jobs according to their periods. If a job has higher frequency they are given more priority. The decision points of this algorithm is the point at which a job arrives in the queue and the point at which a job finishes.

For illustration we will look at the tasks set I used to test the implementation of RMA.

T1: Period 3,execution time 0.5
T2: Period 4, execution time 1.0
T3: Period 6, execution time 2.0

We assume that the deadline of the released jobs is the next release time of the job. The output of the program is as shown below;

In this task set task 3 has the highest period and hence it get’s the highest priority followed by 4 then 6. The jobs are released without any phase difference. The simulation runs for a hyper period, which is the LCM of all the periods. The hyper period is the period at which the schedule repeats itself. So here the hyper period is 12. The format of the output is as follows

Job of task <period of the job> -----> {<entering execution time>   <pre-emption/completion time>

OUTPUT

```
Job of Task 3 ---->{0.000000  0.500000}
Job of Task 4 ---->{0.500000  1.400000}
Job of Task 6 ---->{1.400000  2.999999}
Job of Task 3 ---->{2.999999  3.499999}
Job of Task 6 ---->{3.499999  3.999998}
Job of Task 4 ---->{3.999998  5.999997}
Job of Task 3 ---->{5.999997  6.499996}
Job of Task 6 ---->{6.499996  7.999995}
Job of Task 4 ---->{7.999995  8.899998}
Job of Task 6 ---->{8.899998  8.999998}
Job of Task 3 ---->{8.999998  9.500000}
Job of Task 6 ---->{9.500000  11.900009}
Job of Task 3 ---->{11.900009 ...
```

For convenience of understanding let’s refer to a job using it’s period.

1. Since all the jobs have no phase difference all three jobs are put in the priority queue.
2. Since job 3 has the highest priority it is scheduled first and it executes from 0 to 0.5 and completes it’s execution.
3. Now job 4 enters for execution and executes for 1 unit of time from 0.5 to 1.5.
4. Now after 6 enters the processor it can only execute from time 1.5 to 3.0 since now the second instance of job with period 3 is released. Since the release of a job is a decision point we prempts job 6 and put it back in the priority queue.
5. After 3 executes from 3.0 to 3.5, no other job is ready in the job queue. Hence 6 enters the processor for remaining execution from 3.5 to 4 and completes.
6. At 4 job with period 4 enters the job queue and is taken for execution. It executes from 4 to 5 and it leaves. Now at 6 both job 3 and job 6 third and second instances are released respectively. Since job 3 has the highest priority it is taken for execution over job 6.

ALGORITHM

The C code for this algorithm is given in the appendix.

1. Read task list from file
2. Create a priority queue
3. Find hyper period of tasks using extended Euclid method
4. Initialise the next release of a job to be at the release time
5. To symbolise the processor create a structure variable of type job which stores the parameters of the currently executing job
6. Put all the jobs that have their next execution before the current time into the priority queue.
7. Pop the job at the end of the queue and assign it to now executing
8. initialise idle, interrupt flag to zero
9. System has been initialised
10. Translate through the task array and see if there are any job ready for execution at the given time. If there is a job ready ad it to the priority queue and update the job’s next execution time and set the interrupt flag.
11. If interrupt flag is set and the period of the job at the front of the queue is greater than the job that is currently executing push the job back in the ready queue if the job has not completed, and pop the job at the front of queue and assign it to now executing and reset the interrupt and idle flag
12. If priority queue is not empty and the idle flag is set, pop the job at the front of the queue. Reset interrupt and idle flags
13. decrement the timer of the currently executing job.
14. If the currently running job’s timer has become zero then set the value of the period of now executing job to a high value(greater than the hyper period)
15. increment time
16. if time is lesser than the hyper period go back to step number 10
17. else exit

CHAPTER 3
EARLIEST DEADLINE FIRST

This chapter discusses a popular dynamic priority algorithm called *Earliest deadline first.* The EDF algorithm assigns priorities to individual jobs in a task according to their absolute deadline. As discussed before a task usually comprise of more than one job and a dynamic priority algorithm assigns different priorities to different jobs in the same task.

To illustrate EDF scheduling algorithm let us look at an example of two tasks containing,
T_1={$J_{1,1}$, $J_{1,2}$, $J_{1,3}$... $J_{1,n}$} Period:2 and execution time:0.9
T_2={$J_{2,1}$, $J_{2,2}$,$J_{2,3}$... $J_{2,n}$} Period:5 and execution time:2.3

Let's take the following parameters
1. job $J_{1,1}$ have deadline at 2 and be released at time 0.
2. $J_{1,2}$ have deadline at 5.
3. $J_{1,2}$ have deadline 4 and it is released at time 2.
4. $J_{1,3}$ have deadline 6 and is released at time 4.

The schedule can be written down like the output of the previous program as given the Gantt chart for the same is also shown n figure 1:

$J_{1,1}$ ---->{0.0 0.9}
$J_{2,1}$ ---->{0.9 2.0}
$J_{1,2}$----> {2.0 2.9}
$J_{2,1}$----> {2.9 4.1}
$J_{1,3}$----> {4.1 5.0}

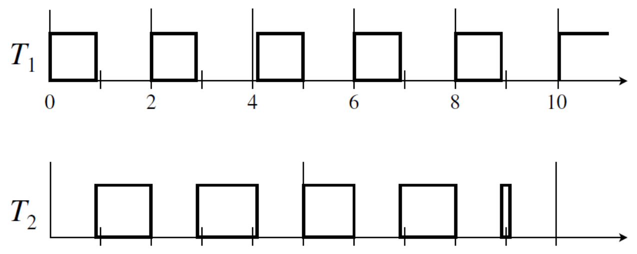

Fig1. *EDF schedule of T_1(2,0.9) and T_2(5,2.3).*

1. Here $J_{1,1}$ has earlier deadline than $J_{2,1}$ hence it enters for execution and completes at 0.9. Then $J_{2,1}$ enters for execution.
2. At time 2 job $J_{1,2}$ is released. It replaces $J_{2,1}$ because it has an earlier deadline and therefore starts execution and completes at 2.9.
3. Then job $J_{2,1}$ comes back for execution.
4. At 4 job $J_{1,3}$ is released but it's deadline is later than $J_{2,1}$ hence $J_{2,1}$ continues it's execution.

CHAPTER 4
SCHEDULING APERIODIC AND SPORADIC TASKS

This chapter deals with the art of scheduling aperiodic and sporadic tasks with periodic tasks. Since sporadic tasks have hard deadlines like periodic tasks we have to be careful about how we place them in the schedule. Since aperiodic tasks doesn't have deadline we just have to be careful about their reaction time, since a late execution of an aperiodic tasks will decrease the quality of the systems performance.

We will also look into an implementation of using *deferred servers* to schedule aperiodic tasks in a real time system. Before we dive into the algorithm we need to state assumptions. Like we assumed the periodic tasks to be independent we also assume that the sporadic and aperiodic tasks are independent. Some sporadic jobs may never meet their deadline. We make the decision of; not executing the sporadic jobs that cannot be scheduled. So altogether we can illustrate how the operating system schedule the three kinds of tasks using different priority queues as shown in fig 3.

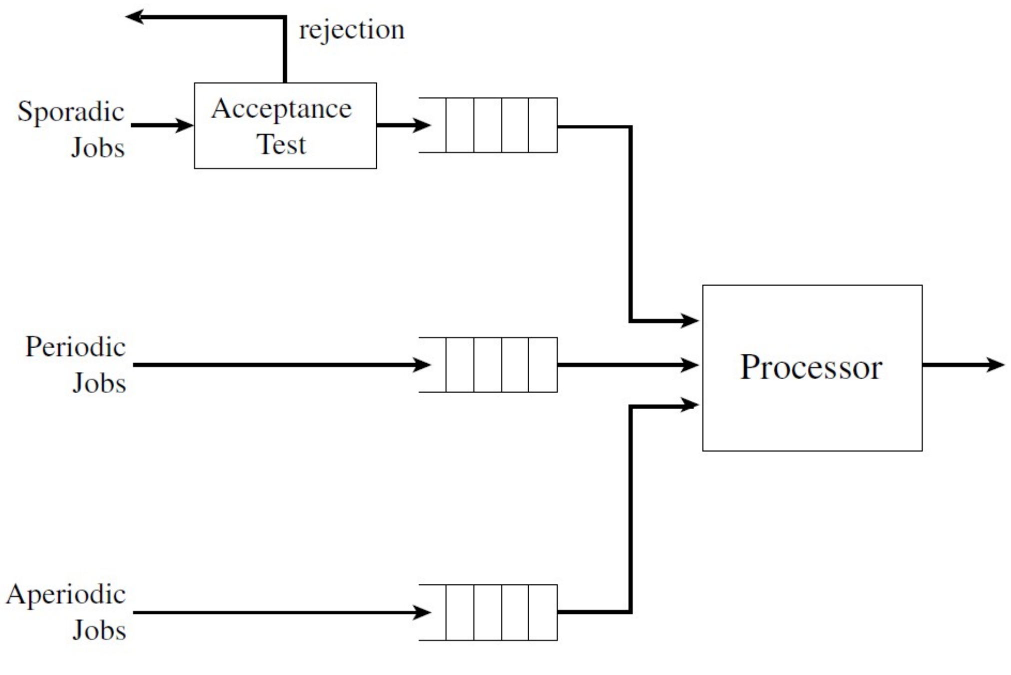

Fig 2.*Priority queues maintained by the operating systems.*

Bandwidth preserving servers

There are many alternative approaches to schedule aperiodic jobs like *background and interrupt driven schedule,slack stealing,polling servers, bandwidth preserving servers etcetera.* We will be focused on scheduling aperiodic jobs using bandwidth preserving servers. We will look into a defenite class of bandwidth reserving server called *deferrable servers.*

A bandwidth preserving server is a method of letting an aperiodic task masquerade as a periodic task so that it can be scheduled using scheduling them using priority-driven algorithms like RMA or EDF. A server is executed in periodic intervals. It has an execution budget in it's every period, at the beginning of every period it checks to see if there are any aperiodic jobs in the aperiodic job queue. If there is a job, it takes that job and executes the job until it's budget is used up. At every start of the period the budget of the server is replenished. If the aperiodic job queue is empty at the time of polling then the server suspends itself and the budget is not exhausted, or if the server completes a job before it's budget is exhausted at the arrival of a new job into the aperiodic job queue the server takes it up and executes it for the remaining time.

For illustration we will take up the output of my implementation of the same in C code(appendix 2):

```
Periodic tasks:
0.000000 3 0.500000 3.000000                //Release time=0, Period=3, Execution time=0.5 and
Deadline=3
0.000000 5 1.000000 4.000000
0.000000 6 2.000000 6.000000

The aperiodic tasks::
3.000000 0.500000                   //Release time=3 Execution time=0.5
5.000000 1.000000

u= 0.700000                         //utilisation of the periodic jobs

Server budget:: 0.100000     Server Period:: 2
Hyperperiod :: 30
Job of Task 3 ---->{ 0.000000  0.500000}
Job of Task 5 ---->{0.500000  1.400000}
Job of Task 6 ---->{1.400000 2.999999}
Aperiodic job {2.999999 3.099999}
Aperiodic job {3.099999 3.199999}
Job of Task 3 ---->{3.199999  3.699999}
Job of Task 6 ---->{3.699999 3.999998}
Aperiodic job {3.999998 4.099998}
Aperiodic job {4.099998 4.199998}
Job of Task 5 ---->{4.999998 5.999997}
Aperiodic job {5.999997 6.099997}
Aperiodic job {6.099997 6.199996}
Job of Task 3 ---->{6.199996  6.699996}
Job of Task 6 ---->{6.699996 7.999995}
```

```
Aperiodic job {7.999995 8.099995}
Aperiodic job {8.099995 8.199995}
Job of Task 3 ---->{8.999998 9.900002}
Aperiodic job {9.900002 10.000002}
Aperiodic job {10.000002 10.100002}
Job of Task 5 ---->{10.100002 11.900009}
Aperiodic job {11.900009 12.000010}
Aperiodic job {12.000010 12.100010}
Job of Task 3 ---->{12.100010  12.600012}
Job of Task 6 ---->{12.600012 13.900017}
Aperiodic job {13.900017 14.000017}
Aperiodic job {14.000017 14.100018}
Job of Task 3 ---->{14.900021  15.400023}
Job of Task 5 ---->{15.400023 15.900024}
Aperiodic job {15.900024 16.000025}
Aperiodic job {16.000025 16.100025}
Aperiodic job {17.900032 18.000032}
Aperiodic job {18.000032 18.100033}
Job of Task 3 ---->{18.100033  18.600035}
Job of Task 6 ---->{18.600035 19.900040}
Aperiodic job {19.900040 20.000040}
Aperiodic job {20.000040 20.100040}
Job of Task 5 ---->{20.100040  20.900043}
Job of Task 3 ---->{20.900043  21.400045}
Job of Task 5 ---->{21.400045  21.500046}
Job of Task 6 ---->{21.500046 21.900047}
Aperiodic job {21.900047 22.000048}
Aperiodic job {22.000048 22.100048}
Aperiodic job {23.900055 24.000055}
Aperiodic job {24.000055 24.100056}
Job of Task 3 ---->{24.100056  24.600058}
Job of Task 6 ---->{24.600058  24.900059}
Job of Task 5 ---->{24.900059  25.800062}
Job of Task 6 ---->{25.800062 25.900063}
Aperiodic job {25.900063 26.000063}
Aperiodic job {26.000063 26.100063}
Job of Task 3 ---->{26.900066  27.400068}
Job of Task 6 ---->{27.400068 27.900070}
Aperiodic job {27.900070 28.000071}
Aperiodic job {28.000071 28.100071}
Aperiodic job {29.900078 30.000078}
...
```

We have determine suitable period and budget of the server. For evaluating this we have to define a few more things. The utilisation of a processor is an important parameter to be calculated before scheduling using a set of tasks on a processor. It is the sum of the ratios of expected execution time by the tasks periods.

$$U_i = \sum_{i}^{n} \left(\frac{e_i}{p_i}\right) \tag{1}$$

The implementation that we are going to discuss uses the RMA schedule. For an RMA schedule to be schedulable we follow the sufficient but not necessary condition known as the Liu-Leyland criterion.

$$U_i \leq n\left(2^{(1/n)} - 1\right) \tag{2}$$

While the necessary condition for an EDF schedule is that the utilisation should be less than 1.

We can use these results to determine the server's period and a suitable budget. First if we calculate the utilisation of the task set without the server, let this be U_i also find the Liu-Leyland bound L_i of this task set. Next calculate the lowest period in the task set and decrease one from it (or any small value, let's deal with integer numbers for simplicity). This is for getting a period which gets highest priority for the server. Now that we have determined a period for the server we can find a suitable budget for the server. Find the utilisation bound L_s when one more task is added to the previous case. Then from equation 1 and 2 we get

$$\frac{e_s}{p_s} = L_s - L_i \tag{3}$$

So now that we have got the period and server we can get into the algorithm is implemented

ALGORITHM

1. Read periodic task and aperiodic task parameters from text file.
2. Find the period and budget of server.
3. Find Hyper period
4. Create a queue for the aperiodic jobs
5. Create a priority queue for the periodic jobs
6. Create an instance of job data type to represent processor.
7. Translate through the periodic task list and insert the job with highest priority according to their period into the priority queue and set the periodic_job_interrupt flag
8. Translate through the aperiodic task list and if a job has been released add it to the queue, and set the aperiodic_job_interrupt flag.
9. Check if the time is equal to a multiple of server period, if so replenish it's budget.

10. Check if server budget is greater than zero and the aperiodic_job_interrupt flag is set ,if so set the server_flag
11. Now if the server flag is set, take the processor and execute the job at the top of the aperiodic job queue push the job that was executing and push it into the periodic job priority queue
12. If interrupt flag is set push the new job into the periodic job queue and take the job at the top of the priority queue and assign it to the processor(the job data type).
13. If the priority queue is not empty and the idle_flag is set then assign the job at the top of the queue to the processor.
14. If server flag is set decrement it's timer(budget) by the set time granularity. Also the timer of the job at the front of the queue.
15. If the timer of the job at the head of the queue has ran out then dequeue that job.
16. If server flag is not set and the decrement the currently executing periodic job timer.
17. If the timer of currently running processor is zero set the idle flag, and the period to a (hyper period+1) to denote a high period to invalidate the job.
18. Increment time by set time granularity
19. If the current time is lesser than the hyper period go back to step 7.

Sporadic job scheduling

The scheduler performs an acceptance test on each sporadic job that is releases. To see if they can be scheduled without disturbing the deadlines of the periodic tasks in the system. Acceptance test take place in EDF order. In a deadline-driven system, they are scheduled with periodic in the EDF order with periodic jobs. Whereas in a fixed priority system they are executed by a bandwidth preserving server. So we shall look into a simple acceptance test that is done on sporadic jobs.

Let's state a theorem without proof: A system of independent, premptable sporadic jobs is schedulable according to the EDF algorithm if the total density of all active jobs in the system is no greater than 1 at all times.

Here density is the sum of $e_i/(d_i-r_i)$ of all jobs where e_i is the execution time, d_i is the deadline of the job and r_i is the release time. So to validate a sporadic job we test if density of all periodic tasks in the system is Δ, then the density of all sporadic jobs is less than $1-\Delta$. For a single sporadic job we just fin the ration of execution time to the difference between deadline and release time. The deadline of this sporadic job divides the schedule into two intervals. The first interval having density equal to Δ_s and the other one having zero. Now extending this to multiple sporadic jobs; a list of n_s jobs will partition the schedule into n_s+1 intervals $I_1,I_2,I_3...I_n$. The scheduler maintains a list of all densities in all these intervals $\Delta_{s,k}$. Now, when a new sporadic job arrives in the queue. The scheduler finds if the following condition holds true

$$\frac{e}{d-t}+\Delta_{s,k}\leq 1-\Delta \qquad (3)$$

for all k=1,2,....l

Where l is the interval in which the deadline of the new sporadic job falls. This new sporadic job makes a new interval. So all the intervals after l is renamed. The ratio e/(d-t) of the new sporadic job is added to density of all intervals below l and this process goes on.

CONCLUSION

The two popular scheduling methods RMA and EDF algorithm was discussed where the fixed priority method (RMA) was implemented. Then it moved on to the discussion of scheduling aperiodic and sporadic jobs with periodic jobs. The RMA implementation was extended to include aperiodic jobs, where the aperiodic jobs were scheduled using deferred servers. The implementation and study gave insight into the nuts and bolts of real time operating systems.

REFERENCE

[1] Liu , Jane W .S, *Real-time systems,* Prentice hall ,2000

List of figures

APPENDIX-1 – Implementation of RMA scheduler

Header file sched.h

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

#define TIME_GRAN 0.1
#define READY_QUEUE_SIZE 20 //maximum nummber of tasks possible to resife on
main memory
#define RUNNING_TIME 4

typedef struct{
	float next_exec;
	int period;	//if period is -1 then it is an aperiodic process
	float exec;
	float release_time;	//if it is -1 then this task  instance does not
exist(error  checking in queue
	float deadline;	//if deadline is -1 then it is an aperiodic task
	int id;
}task;

typedef struct
{
	int front, rear, size;
	unsigned capacity;
	task* array;
}TQueue;

typedef struct{
	float timer;
	float exec;
	float period;
	float deadline;
}server;

typedef struct Job{
	int id;
	float timer;
	int period;
	struct Job* next;
}job;


job* newJob(float d, int p) ;
```

```
int peekp(job** head);
float peekx(job** head);
void pop(job** head) ;
void push(job** head, float d, int p) ;
int isEmpty(job** head) ;


int create_queue(FILE* fp,TQueue* tq);
void rms_schedule(task* ptasks,int no_of_ptasks);
```

Data structure file structs.c

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

job* newJob(float d, int p)
{
    job* temp = (job*)malloc(sizeof(job));
    temp->timer= d;
    temp->period = p;
    temp->next = NULL;
    return temp;
}


int peekp(job** head)
{
    return (*head)->period;
}

float peekx(job** head)
{
    return (*head)->timer;
}

void pop(job** head)
{
    job* temp = *head;
    (*head)= (*head)->next;
    free(temp);
}
int peeki(job** head){
      return (*head)->id;
}
```

```
void push(job** head, float d, int p)
{
    job* start = (*head);

    job* temp = newJob(d, p);

    if ((*head)->period > p) {
        temp->next = *head;
        (*head) = temp;
    }
    else {


        while (start->next != NULL &&
            (start->next)->period <p) {
            start = start->next;
        }

        temp->next = start->next;
        start->next = temp;
    }
}


int isEmpty(job** head)
{
    return (*head) == NULL;
}
```

<u>RMA_Sched.c</u>

```
#include "sched.h"
#include<stdio.h>
#include<stdlib.h>
#include<strings.h>
#include<math.h>
int lcm(int a, int b)
{
    int m = 1;

    while(m%a || m%b) m++;

    return m;
}


void rms_schedule(task* ptasks,int no_of_ptasks){
```

```
float t=0;       //time set to zero
job* job_q;
int hyperperiod=ptasks[0].period;
for(int i=0;i<no_of_ptasks;i++)
      hyperperiod=lcm(hyperperiod,ptasks[i].period);
printf("Hyperperiod :: %d\n",hyperperiod);
for(int i=0;i<no_of_ptasks;i++)
      ptasks[i].next_exec=ptasks[i].release_time;
int flag=0;
job now_executing;
for(int i=0;i<no_of_ptasks;i++){
      float dif=ptasks[i].next_exec-t;
      if(dif<TIME_GRAN && flag==0){
             job_q=newJob(ptasks[i].exec,ptasks[i].period);
             ptasks[i].next_exec+=ptasks[i].period;
             now_executing.period=peekp(&job_q);
             now_executing.timer=peekx(&job_q);
             flag=1;
      }
      else if(dif<TIME_GRAN && flag==1){
             push(&job_q,ptasks[i].exec,ptasks[i].period);
              ptasks[i].next_exec+=ptasks[i].period;
              if(peekp(&job_q)<now_executing.period){
                  now_executing.period=peekp(&job_q);
                  now_executing.timer=peekx(&job_q);

             }
      }
}
now_executing.next=NULL;
pop(&job_q);
//printf("%d %f\n",now_executing.period,now_executing.timer);
/* while (!isEmpty(&job_q)) {
      printf("%d \n", peekp(&job_q));
      pop(&job_q);
 } */

int idle=0,intr=0;
int print_f=0;
printf("Job of Task %d ---->{ %f ",now_executing.period,t);
while(t<=hyperperiod){
      for(int i=0;i<no_of_ptasks;i++){
            float dif=ptasks[i].next_exec-t;
            if(dif<0.1){
                  //printf("dif:: %f\n",dif);
                  if(!isEmpty(&job_q))
                        push(&job_q,ptasks[i].exec,ptasks[i].period);
                  else
                        job_q=newJob(ptasks[i].exec,ptasks[i].period);
                  //printf("%d \n", peekp(&job_q));
                  ptasks[i].next_exec+=ptasks[i].period;
                  intr=1;
```

```
                 }
            }
            if(intr==1  && peekp(&job_q)<now_executing.period){
                  printf(" %f}\n",t);
                  if(now_executing.timer>=TIME_GRAN){

      push(&job_q,now_executing.timer,now_executing.period);

                  }
                  now_executing.timer=peekx(&job_q);
                  now_executing.period=peekp(&job_q);
                  pop(&job_q);

                  printf("Job of Task %d ---->{%f ",now_executing.period,t);
                  idle=0;
                  intr=0;
            }
            if(!isEmpty(&job_q) && idle==1){
                  printf(" %f}\n",t);
                  now_executing.timer=peekx(&job_q);
                  now_executing.period=peekp(&job_q);
                  pop(&job_q);
                  intr=0;
                  idle=0;
                  printf("Job of Task %d ---->{%f ",now_executing.period,t);
            }

            //printf("\n%f\n",now_executing.timer);
            now_executing.timer-=0.1;
            if(now_executing.timer<TIME_GRAN){
                  idle=1;
                  now_executing.period=100000;
            }
            t=t+TIME_GRAN;

      }
      printf("...\n");

}
```

<u>DRIVER PROGRAM</u>

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

```
#include"sched.h"

int main(int argc,char* argv[]){
	float t=0;
	FILE* fp;
	fp=fopen("periodic.txt","rw");
	int no_of_ptasks;
	fscanf(fp,"%d",&no_of_ptasks);
	task* ptasks=(task*)malloc(sizeof(TQueue)*no_of_ptasks);
	for(int i=0;i<no_of_ptasks;i++){
		fscanf(fp,"%f %d %f %f
%d",&ptasks[i].release_time,&ptasks[i].period,&ptasks[i].exec,&ptasks[i].dea
dline,&ptasks[i].id);
		printf("%f %d %f %f\
n",ptasks[i].release_time,ptasks[i].period,ptasks[i].exec,ptasks[i].deadline
);
	}
	rms_schedule(ptasks,no_of_ptasks);
	return 0;
}
```

APPENDIX 2 -Modified scheduler to include aperiodic task scheduling

Functions used for scheduling

```
#include "sched.h"
#include<stdio.h>
#include<stdlib.h>
#include<strings.h>
#include<math.h>

void initialise_server(task* ptasks,int no_of_ptasks,server* serv){
	int lst;
	for(int i=0;i<no_of_ptasks;i++)
		if(ptasks[i].period<lst)
			lst=ptasks[i].period;

	float ui,us,limit;
	ui=0;
	for(int i=0;i<no_of_ptasks;i++)
		ui=ui+(ptasks[i].exec/ptasks[i].period);
	printf("ui= %f\n",ui);
	limit=(no_of_ptasks+1)*(pow(2,1/((float)no_of_ptasks+1))-1)-ui;
	printf("%f\n",limit);
	serv->period=lst-1;
	float b=0;
	while(1){
		b=b+0.1;
		if((b/serv->period)>=limit)
			break;
	}
	serv->budget=b-0.1;
	printf("Server budget:: %f Server Period:: %d\n",serv->budget,serv-
>period);
	serv->next_exec=0;
	serv->timer=serv->budget;
}

int lcm(int a, int b)
{
    int m = 1;

    while(m%a || m%b) m++;

    return m;
}
```

```
void rms_schedule(task* ptasks,task* atasks,int no_of_ptasks,int
no_of_atasks){
	float t=0;	//time set to zero
	job* job_q;
	server* serv=(server*)malloc(sizeof(server)*1);
	initialise_server(ptasks,no_of_ptasks,serv);
	int hyperperiod=ptasks[0].period;
	for(int i=0;i<no_of_ptasks;i++)
		hyperperiod=lcm(hyperperiod,ptasks[i].period);
	hyperperiod=lcm(hyperperiod,serv->period);
	printf("Hyperperiod :: %d\n",hyperperiod);
	for(int i=0;i<no_of_ptasks;i++)
		ptasks[i].next_exec=ptasks[i].release_time;
	int flag=0;
	job now_executing;
	AQueue* aq=createQueue(5);
	/*for(int i=0;i<no_of_atasks;i++){
		ajob temp;
		temp.timer=atasks[i].exec;
		temp.release_time=atasks[i].release_time;
		enqueue(aq,temp);		//aperiodic task queue has been
created
	}*/
	//print_queue(aq);
	for(int i=0;i<no_of_ptasks;i++){
		float dif=ptasks[i].next_exec-t;
		if(dif<TIME_GRAN && flag==0){
			job_q=newJob(ptasks[i].exec,ptasks[i].period);
			ptasks[i].next_exec+=ptasks[i].period;
			now_executing.period=peekp(&job_q);
			now_executing.timer=peekx(&job_q);
			flag=1;
		}
		else if(dif<TIME_GRAN && flag==1){
			push(&job_q,ptasks[i].exec,ptasks[i].period);
			ptasks[i].next_exec+=ptasks[i].period;
			if(peekp(&job_q)<now_executing.period){
				now_executing.period=peekp(&job_q);

				now_executing.timer=peekx(&job_q);

			 }
		}
	}
	now_executing.next=NULL;
	pop(&job_q);
```

```
      int hijack=0,intra=0,ap_f=0,server_f=0;//these falgs are required
to shift the processor from old rma to the one with servers
      int idle=0,intr=0;
      int print_f=0;
      printf("Job of Task %d ---->{ %f ",now_executing.period,t);
      while(t<=hyperperiod){
            for(int i=0;i<no_of_ptasks;i++){
                  float dif=ptasks[i].next_exec-t;
                  if(dif<0.1){
                        if(!isEmpty(&job_q))
                              push(&job_q,ptasks[i].exec,ptasks[i].period);
                        else
                              job_q=newJob(ptasks[i].exec,ptasks[i].period);
                        ptasks[i].next_exec+=ptasks[i].period;
                        intr=1;
                   }
            }
            for(int i=0;i<no_of_atasks;i++){
                  if(atasks[i].release_time-t<0.1){
                        ajob temp;
                        temp.timer=atasks[i].exec;
                        temp.release_time=atasks[i].release_time;
                        enqueue(aq,temp);           //aperiodic task queue
has been created
                        intra=1;
                  }

            }

            if( serv->next_exec-t<0.1){
                  serv->next_exec+=serv->period;
                  serv->timer=serv->budget;
                  if(!qisEmpty(aq))
                        server_f=1;
            }
            else if(intra==1 && serv->timer>0)
                  server_f=1;

            if(server_f==1){
                  printf("%f}\n",t);
                  hijack=1;
                  printf("Aperiodic job {%f %f}\n",t,t+serv->budget);
                  ap_f=1;
                  server_f=0;
            }

            else if(intr==1  && peekp(&job_q)<now_executing.period){
//decision point at the arrival of a job
                  if(hijack==0)
```

```
                        printf(" %f}\n",t);  //hijack flag
                    else
                        hijack=0;
                    if(now_executing.timer>=TIME_GRAN){

            push(&job_q,now_executing.timer,now_executing.period);

                    }
                    now_executing.timer=peekx(&job_q);
                    now_executing.period=peekp(&job_q);
                    pop(&job_q);

                    printf("Job of Task %d ---->{%f
",now_executing.period,t);
                    idle=0;
                    intr=0;
                }
                else if(!isEmpty(&job_q) && idle==1){ //decision point at
which a job leaves at t-0.1
                    if(hijack==0)
                        printf(" %f}\n",t);  //hijack flag
                    else
                        hijack=0;
                    now_executing.timer=peekx(&job_q);
                    now_executing.period=peekp(&job_q);
                    pop(&job_q);
                    intr=0;
                    idle=0;
                    printf("Job of Task %d ---->{%f
",now_executing.period,t);
                }
                  if(ap_f==0){
                    now_executing.timer-=0.1;
                }
                else if(ap_f==1){
                    serv->timer-=0.1;
                    aq->array[aq->front].timer-=0.1;
                    if(aq->array[aq->front].timer <=0){
                        dequeue(aq);
                        ap_f=0;
                    }
                    if(serv->timer<=0){
                        ap_f=0;
                    }
                }

                if(now_executing.timer<TIME_GRAN && ap_f==0){
                    idle=1;
                    now_executing.period=100000;
```

```
		}
		t=t+TIME_GRAN;

	}
	printf("...\n");

}
```

Header file containing definitions of structures

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

#define TIME_GRAN 0.1
#define READY_QUEUE_SIZE 20     //maximum nummber of tasks possible to
resife on main memory
#define RUNNING_TIME 4

typedef struct{
	float next_exec;
	int period;		//if period is -1 then it is an aperiodic
process
	float exec;
	float release_time;	//if it is -1 then this task  instance does
not exist(error  checking in queue
	float deadline;	//if deadline is -1 then it is an aperiodic
task
	int id;
}task;



typedef struct{
	float budget;
	int period;
	float next_exec;
	float timer;
}server;

typedef struct Job{
	int id;
	float timer;
	int period;
	struct Job* next;
}job;
typedef struct{
	float timer;
	float release_time;
}ajob;

typedef struct
{
	int front, rear, size;
	unsigned capacity;
	ajob* array;
}AQueue;



job* newJob(float d, int p) ;
int peekp(job** head);
float peekx(job** head);
void pop(job** head) ;
void push(job** head, float d, int p) ;
int isEmpty(job** head) ;

//queue functions

AQueue* createQueue(unsigned capacity) ;
int qisFull(AQueue* queue);
int qisEmpty(AQueue* queue);
void enqueue(AQueue* queue, ajob item);
ajob dequeue(AQueue* queue) ;
ajob front(AQueue* queue);
ajob rear(AQueue* queue);
void print_queue(AQueue* queue);

//scheduling functions
int create_queue(FILE* fp,AQueue* tq);
void rms_schedule(task* ptasks,task* atasks,int no_of_ptasks,int
no_of_atasks);
void initialise_server(task* ptasks,int no_of_ptasks,server* serv);
```

Data structures used for deffered servers and RMA scheduling

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

job* newJob(float d, int p)
{
    job* temp = (job*)malloc(sizeof(job));
    temp->timer= d;
    temp->period = p;
    temp->next = NULL;
    return temp;
}


int peekp(job** head)
{
    return (*head)->period;
}

float peekx(job** head)
{
    return (*head)->timer;
}

void pop(job** head)
{
    job* temp = *head;
    (*head)= (*head)->next;
    free(temp);
}
int peeki(job** head){
     return (*head)->id;
}


void push(job** head, float d, int p)
{
    job* start = (*head);

    job* temp = newJob(d, p);

    if ((*head)->period > p) {
        temp->next = *head;
        (*head) = temp;
    }
    else {


        while (start->next != NULL &&
            (start->next)->period <p) {
            start = start->next;
        }

        temp->next = start->next;
        start->next = temp;
    }
}


int isEmpty(job** head)
{
    return (*head) == NULL;
}


/
************************************************************************
********************************************************************/
// function to create a queue of given capacity.
// It initializes size of queue as 0
AQueue* createQueue(unsigned capacity) {
    AQueue* queue = (AQueue*) malloc(sizeof(AQueue));
    queue->capacity = capacity;
    queue->front = queue->size = 0;
    queue->rear = capacity - 1;  // This is important, see the enqueue
    queue->array = (ajob*) malloc(queue->capacity * sizeof(ajob));
    return queue;
}

// Queue is full when size becomes equal to the capacity
int qisFull(AQueue* queue)
{  return (queue->size == queue->capacity);  }

// Queue is empty when size is 0
int qisEmpty(AQueue* queue)
{  return (queue->size == 0); }

// Function to add an item to the queue.
// It changes rear and size
void enqueue(AQueue* queue, ajob item)
{
    if (qisFull(queue))
```

```
        return;
    queue->rear = (queue->rear + 1)%queue->capacity;
    queue->array[queue->rear] = item;
    queue->size = queue->size + 1;
    //printf("Task  enqueued to queue\n");
}

// Function to remove an item from queue.
// It changes front and size
ajob dequeue(AQueue* queue) {
    if (qisEmpty(queue)){
         ajob temp={-1,0};
         return temp;
    }
    ajob item = queue->array[queue->front];
    queue->front = (queue->front + 1)%queue->capacity;
    queue->size = queue->size - 1;
    return item;
}

// Function to get front of queue
ajob front(AQueue* queue) {
    if (qisEmpty(queue)) {
       ajob temp={-1,0};
         return temp;
       }

   return queue->array[queue->front];
}

// Function to get rear of queue
ajob rear(AQueue* queue) {
    if (qisEmpty(queue)) {
         ajob temp={-1,0};
         return temp;
    }
    return queue->array[queue->rear];
}

void print_queue(AQueue* tq){         //used for debugging only  WARNING!
: empties queue
     ajob read_p;
     int n=tq->size;//because each time you dequeue the value of tq-
>size changes
     for(int i=0;i<n;i++){
          read_p=dequeue(tq);
          printf("Queue items %f %f\
n",read_p.timer,read_p.release_time);
```

```
	}
}
```

DRIVER PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<string.h>
#include"sched.h"

int main(int argc,char* argv[]){
	float t=0;
	FILE* fp;
	fp=fopen("periodic.txt","rw");
	int no_of_ptasks;
	fscanf(fp,"%d",&no_of_ptasks);
	task* ptasks=(task*)malloc(sizeof(task)*no_of_ptasks);
	for(int i=0;i<no_of_ptasks;i++){
		fscanf(fp,"%f %d %f %f
%d",&ptasks[i].release_time,&ptasks[i].period,&ptasks[i].exec,&ptasks[i].
deadline,&ptasks[i].id);
		printf("%f %d %f %f\
n",ptasks[i].release_time,ptasks[i].period,ptasks[i].exec,ptasks[i].deadl
ine);
	}
	FILE* fpa=fopen("aperiodic.txt","rw");
	int no_of_atasks;
	fscanf(fpa,"%d",&no_of_atasks);
	task* atasks=(task*)malloc(sizeof(task)*no_of_atasks);
	printf("The aperiodic tasks:: \n");
	for(int i=0;i<no_of_atasks;i++){
		fscanf(fpa,"%f %f",&atasks[i].release_time,&atasks[i].exec);

		printf("%f %f\n",atasks[i].release_time,atasks[i].exec);
	}

	rms_schedule(ptasks,atasks,no_of_ptasks,no_of_atasks);
	return 0;
}
```