Ain shams university

Faculty of engineering

Computer engineering and software systems programs

Course: **Real Time and Embedded Systems Design** (**CSE 345**)

**Project 1 Team Report**

Submitted by:

Shehab Ahmed Hassan Kotb 16p6014

Basil Essam Abd El Fattah 16p6013

Hazem Hamada Abdellatif 16p3100

Gasser Tarek Ali 16p6024

Submitted to: Dr. Omer Al Kelany

Eng. Mostafa Arafa

Table of Contents

[1.0 System Design 3](#_Toc41458599)

[1.1 Project Description 3](#_Toc41458600)

[1.2 Project Specification 3](#_Toc41458601)

[1.3 Design Choices 4](#_Toc41458602)

[1.4 Block Diagram/FlowChart 5](#_Toc41458603)

[1.5 Team Responsibilities 6](#_Toc41458604)

[1.6 Plan of TimeLine 7](#_Toc41458605)

[2.0 Technical Details 8](#_Toc41458606)

[2.1 Design Details 8](#_Toc41458607)

[2.2 Test Scenarios 8](#_Toc41458608)

[2.3 Code Listing 9](#_Toc41458609)

[2.3.1 Data Types 9](#_Toc41458610)

[2.3.2 Functions 10](#_Toc41458611)

[2.4 Lessons Learned 13](#_Toc41458612)

[2.5 Problems Faced 13](#_Toc41458613)

[3.0 Research 14](#_Toc41458614)

[3.1 Introduction 14](#_Toc41458615)

[3.2 Rate Monotonic Scheduling 15](#_Toc41458616)

[3.3 Utilization 18](#_Toc41458617)

[3.4 Conclusion 19](#_Toc41458618)

[References 20](#_Toc41458619)

# 1.0 System Design

## 1.1 Project Description

The goal of the project is to extend the FreeRTOS scheduler to make it capable of doing rate monotonic scheduling, we generate the variables for each task before we begin the scheduler as this is a prototype, variables generated are things like task period, arrival time, execution time, and deletion time. Our rate monotonic scheduler takes these variables and decides if the tasks can be scheduled, it also assigns priorities to tasks, and handles late arrival, deletion of task randomly.

## 1.2 Project Specification

1. **Calculating CPU utilization percentage:** if the utilization of the generated task is less than 70% we continue on to scheduling if it exceeds that number we don’t schedule
2. **Setting tasks priorities:** if the tasks are schedulable, we assign the task priority based on all tasks period in descending fashion.
3. **Handling late task arrival time:** if a task arrival time is bigger than 0 we should not take it into consideration when we start the schedular, only is it taken into consideration when its arrival time comes.
4. **Handling random task deletion:** the schedular should also handle random deletion time for random tasks.
5. **Different modes for period generation:** there is a **safe mode** for period generation which is done with the following formula **3xTc(i) to maximum\_period\_multipler x Tc(i)**, also there is **no guarantee mode** calculated with the following formula from 3xTc(i) to 10xTc(i).

## 1.3 Design Choices

**Generating all Tasks Parameters at The Start Including Deletion Time**

For the sake of simplicity and design we decided to generate a delete for each task to be handled in similar fashion of how arrival time is handled.

**Creating the RMS schedular as a master task**

We created a task that acts as the rate monotonic schedular that has the highest priority of all tasks and runs periodically every time slice and it checks the event queues for arrival and deletion of tasks and handles each of these events in a similar respectively.

**Using a dynamic data structure for tasks data**

We used a dynamically allocated array of structs of type **task\_data** which contains all task parameters along with the **xTaskHandle** of that task.

**Using event queues to schedule arrival and deletion of tasks**

After we generate all parameters of tasks including arrival and deletion time at the start, we sort the data by arrival time in ascending order, then we add each task that has arrival time bigger than 0 to our arrival event queue.

Similarly, we do this with deletion time we sort by the tasks delete time in ascending order and then add them to the deletion event queue.

What we add to the queues in both cases is a struct called **Task\_Event** which **contains** the **time of event** and **type of event** and also all task parameter including in its task handler.

When the our monotonic schedular starts it interrupts the whatever task is executing every second to check the event queue, if current time slice is equal or greater than time of first items in the queue, we dequeue and handle it appropriately whether it’s a deletion or arrival event.

## 1.4 Block Diagram/FlowChart

This is a flow chart of the main function; it doesn’t follow the code implementation exactly it acts more of a pseudocode of what happens in the main function.

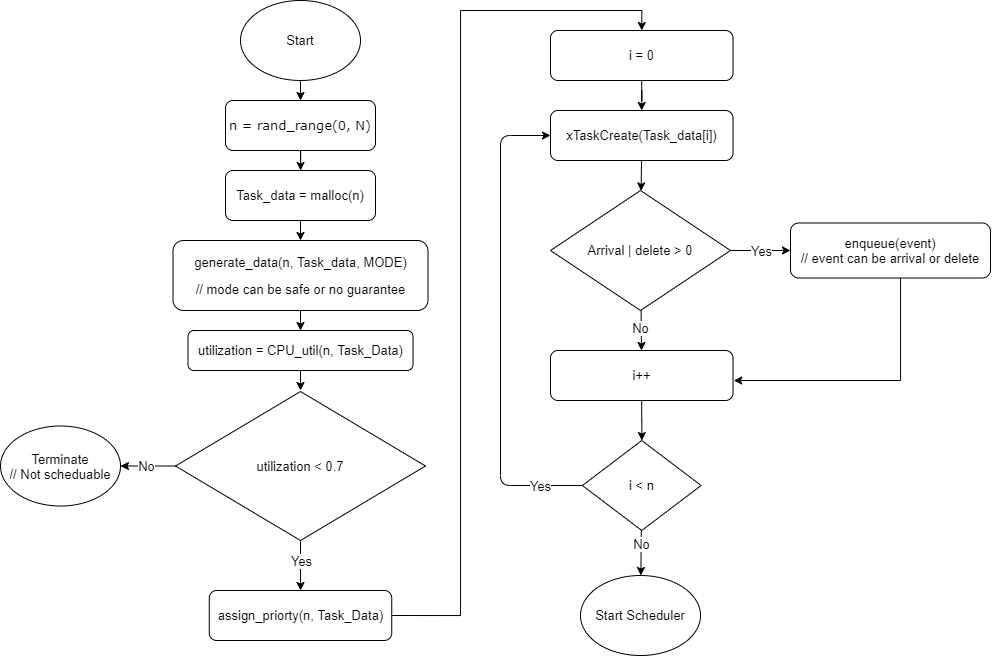


figure - Program Flow Chart

## 1.5 Team Responsibilities

**Basil**

1. Was responsible for the initialization of port A and UART0
2. Was responsible for the setup of TivaWare library and also included the utils Tivaware library which contained the UARTprintf() functions.

**Gasser**

1. Was responsible for the **research portion** in this project.
2. Also created the function that its used to calculate the utilization of the tasks.

**Hazem**

1. Was responsible for generating the tasks data/parameters
2. Also created the random task function that is used to simulate tasks executing.

**Shehab**

1. Created the void pointer queue implementation to be used with task events.
2. Built all the logic for queuing all the arrival and deletion events in the main function.
3. Wrote all the logic in the monotonic schedular task to handle the arrival and deletion of tasks.

**Table of functions and their respective developer**

|  |  |
| --- | --- |
| Developer | Functions/Responsibilities |
| Basil | void print\_data(task\_data\* x);  void print\_data\_array(task\_data\*\* x, int n);  void print\_task\_list(void); |
| Gasser | float cpu\_utilization(unsigned int n, task\_data\*\* data);  Research Portion |
| Hazem | void generate\_task\_data(unsigned n, task\_data\*\* data, bool safe\_mode);  void assign\_priorty(unsigned int n, task\_data\*\* data);  static void random\_task(void \*pvParameters); |
| Shehab | My\_queue.h & My\_queue.c Void pointer queue implementation  void monotonic\_scheduler(void \*pvParameters);  void process\_event(Task\_Event\* event);  void delete\_task\_data(task\_data\* task, task\_data\*\* task\_array, unsigned n);  void fix\_task\_priorities(task\_data\*\* task\_array, unsigned int n);  Event queues logic in main function |

## 1.6 Plan of TimeLine

|  |  |
| --- | --- |
| Deadline Dates | Goals |
| Tue 5/5 | Installing TivaWare for UART functions |
| Thru 14/5 | Designing the scheduler |
| Mon 18/5 | Writing all the code |
| Sun 24/5 | Finishing up the reports |

# 2.0 Technical Details

## 2.1 Design Details

**Using Busy Wait to Simulate Computation Time**

Since tasks here are just simulated and not real tasks we use a busy wait function for the computation duration of the task, also this makes it realistic when another task interrupts, when a task return from the interruption it will still have to complete the rest of the busy wait loop.

**Generating the Deletion Times at the Beginning**

For the sake of simplicity and to make dealing with deletion similar to how we deal with arrival times, we decided to generate them at the beginning and add an event for each deletion in time that our monotonic schedular handles when the time slice comes.

The formula with which we generate deletion time is dependant on the period of the task  
**Ta(i) + 2\*Tc(i) to Ta(i) + 6\*Tc(i);** where Ta, Tc are arrival and computation time respectively.

**Using TivaWare Utility Library**

TivaWare library has a utility library which has an implementation for **printf** for the UART called **UARTprintf** which saves a lot of time instead of having to implement our own UART printing functions.

**Using queues to Handle Task Arrival and Delete Events**

The way we handle arrival of task and the deletion of them is in a very similar fashion, when we create the tasks in the main function using **xTaskCreate** function, each arrival and delete time gets enqueued in their respective event queue and then passed to our schedular task, which handles them when their time slice comes.

## 2.2 Test Scenarios

**Input Parameters:**

**N:** 5

**TST:** 1

**LATEST\_ARRIVAL\_TIME** 15

**MAXIMUM\_COMPUTATION\_TIME** 8

**MAXIMUM\_PERIOD\_MULTIPLER** 17

**MODE:** NO\_GUARANTEE\_MODE

**Output:**

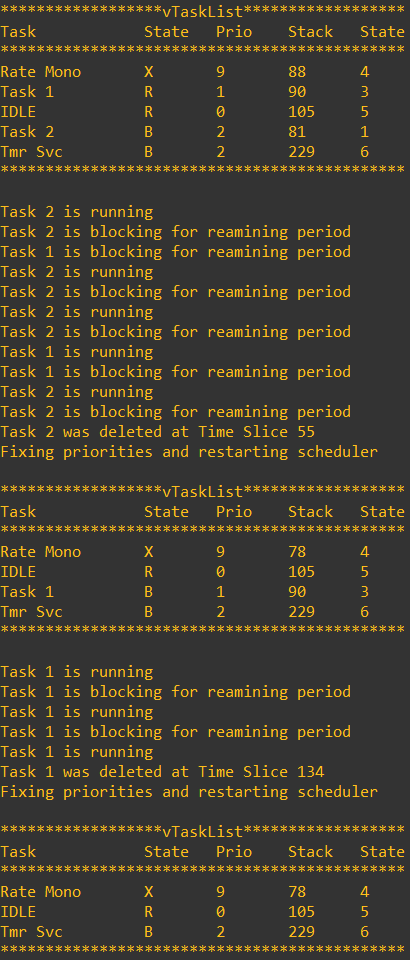
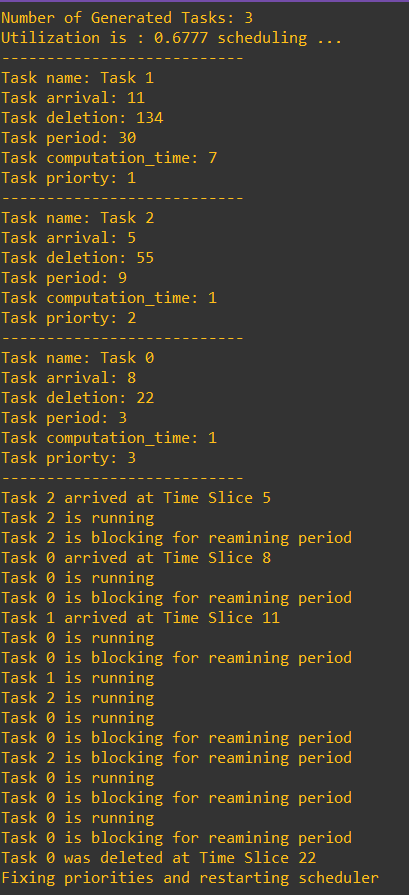
****

figure – Output figure 3 - Output Cont.

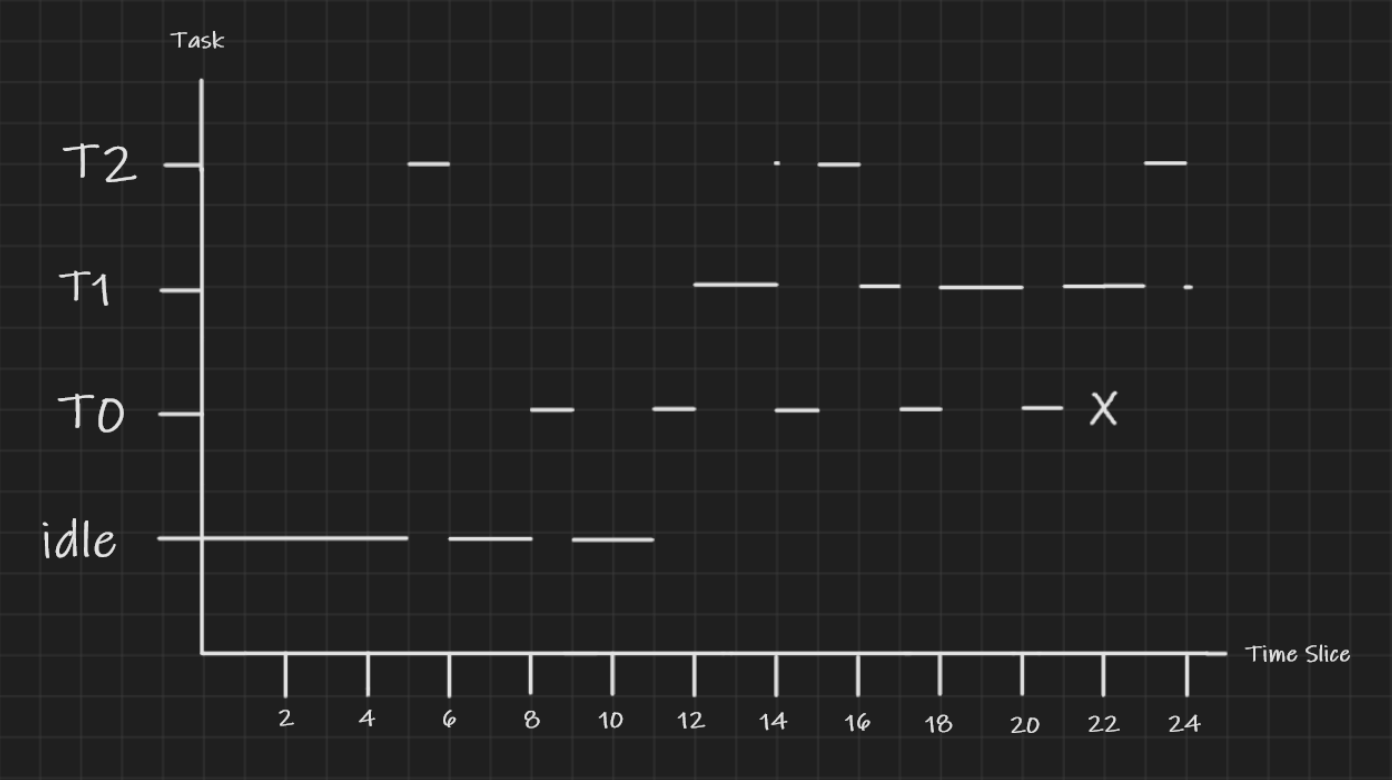


figure - Time Line

**Explanation**

## 2.3 Code Listing

### 2.3.1 Data Types

**struct task\_data**

A struct that contains all task parameters

* unsigned int arrival\_time;
* unsigned int period;
* unsigned int computation\_time;
* unsigned int priorty;
* unsigned int delete\_time;
* TaskHandle\_t handler;
* char\* name;

**struct Task\_Event**

A struct used to schedule arrival and delete events.

* unsigned int event\_time;
* enum Event\_Type type;
* task\_data\* task;

**enum Event\_Type** can take the following values: **ARRIVAL, DELETE.**

**struct Event\_Queues**

Used to wrap arrival, deletion queues and task data to pass to our monotonic schedular task.

* **struct My\_queue\* arrival\_events;**
* **struct My\_queue\* delete\_events;**
* **task\_data\*\* data;**

### 2.3.2 Functions

**unsigned int rand\_range(unsigned int lower, unsigned int upper)**

A function to generate random integers between the range lower to upper.

**void generate\_task\_data(unsigned n, task\_data\*\* data, bool safe\_mode)**

This function used to generate all task parameters, there is also a choice between safe mode and no guarantee mode for the task period generation.

**float cpu\_utilization(unsigned int n, task\_data\*\* data)**

This function returns the CPU utilization percentage of the tasks that were generated.

**void assign\_priorty(unsigned int n, task\_data\*\* data)**

This task assigns priority for the tasks data in increasing fashion according to the highest task period to the lowest one.

**void monotonic\_scheduler(void \*pvParameters)**

The task function for our monotonic scheduler it runs periodically every time slice and checks for events that arrived or needs to be deleted, we pass to it **Event\_Queues** through **pvParameters**.

**void process\_event(Task\_Event\* event)**

This function is called from inside our monotonic scheduler it resumes tasks if its arrival rime has come, or deletes a task if its deletion time has come, and prints through the UART the appropriate message.

**void delete\_task\_data(task\_data\* task, task\_data\*\* task\_array, unsigned n)**

This function is also called from monotonic scheduler, it clears the task data from the **task\_array**, which contains all our tasks data.

**void fix\_task\_priorities(task\_data\*\* task\_array, unsigned int n)**

This function is used to change the tasks priorities after a task has been deleted.

**static void random\_task(void \*pvParameters)**

This is the task function used for all generated tasks, we pass through the **pvParammeters** the task data needed, the function contains a busy waiting loop acting as a place holder for computation time, the function also uses **vTaskDelayUntil**() to block when it finishes execution for its remaining period.

**Utility Functions**

**void wait\_1ms(unsigned long msec)**

Used to simulate execution time in a random task.

**int cmp\_period(const void\* a, const void\* b)**

**int cmp\_arrival(const void\* a, const void\* b)**

**int cmp\_delete(const void\* a, const void\* b)**

These three functions are passed to quick sort function in c to sort the task data according to period, arrival and deletion time.

**void print\_data(task\_data\* x)**

**void print\_data\_array(task\_data\*\* x, int n)**

**void print\_task\_list(void)**

These three function are used for printing to the UART, they use functions in the TivaWare library.

**void uart\_init(void)**

**void ports\_init(void)**

Used to initialize port A and UART0 for printing.

**My\_queue.h**

File that includes the implementation of a queue that works on void pointers, so we can queue any type of struct we want, it’s used to create the arrival and deletion event queues.

Functions in the implementation:

* **struct My\_queue\* createQueue(unsigned capacity);**
* **int isFull(struct My\_queue\* queue)**
* **int isEmpty(struct My\_queue\* queue)**
* **void enqueue(struct My\_queue\* queue, void\* item)**
* **void\* dequeue(struct My\_queue\* queue)**
* **void\* front(struct My\_queue\* queue)**
* **void\* rear(struct My\_queue\* queue)**

## 2.4 Lessons Learned

1. we learned how to combine the TivaWare library and FreeRtos and use both of them to shorten development time if the functionality included them helps our project.
2. We learned how to create our own data structures with void pointers to achieve great flexibility.
3. We deepened our understanding of how schedulers work in general since what we created is similar to an extension to the FreeRtos schedular, by allowing to run in a different schedule mode than the default.

## 2.5 Problems Faced

1. Designing a schedular to handle late task arrival and random deletion was a challenge in design that led us to create our own implementation of a queue with void pointers to create arrival, deletion events queues to handle each of them respectively in our schedular.
2. Dealing with dynamically allocated data was also challenging to debug on keil since its debug/watch windows didn’t show pointer arrays easily, which made it harder to debug.

# 3.0 Research

## 3.1 Introduction

In a real-time system, the scheduling algorithm determines the order in which the tasks are executed and the amount of time allowed for each task in the system so that no task (for hard real-time systems) or a minimum number of tasks (for soft real-time systems) misses their deadlines.

**The scheduling algorithm** ensures that critical timing constraints, such as deadlines and response time, are met. If needed, decisions are made that favor the most critical timing constraints, even at the cost of violating others. In soft real-time embedded systems, the real-time scheduling is also used to allocate processor time between tasks.

Real-time systems designers use different schedulability conditions or known as schedulability tests to confirm if a scheduler ensures the fulfillment of constraints of a task set. The condition of schedulability indicates whether a given task set can be scheduled with a given algorithm of scheduling so none of the tasks in the set miss their deadlines.

Many real-time systems use preemptive multitasking, especially those with an underlying real-time operating system (RTOS). **Preemptive multitasking** is a task in which a computer operating system uses certain parameters to determine how long it would take to assign to any one process before allowing another task a turn to use the operating system.

Tasks are assigned priorities, and the RTOS always executes the ready task with highest priority. There are few classes of priorities such as fixed priority, dynamic priority and mixed priority. Fixed priority algorithms are usually simpler than algorithms that needs to computers to calculate priorities while running programs.

One can classify the scheduling algorithms as static and dynamic. “A priori” all scheduling decisions are made in a static scheduling algorithm. A table showing the beginning and completion times of each task is built for a given set of timing constraints, so no task misses its deadline. This method is highly predictable but if the mission parameters change, The table must be recalculated and the system rebooted.

The scheduling decisions are made at run-time in dynamic scheduling algorithms based on the priorities of the tasks. These priority values are used to determine the order of the tasks to be performed. Depending on the dynamic scheduling algorithm priority values may be allocated statically or dynamically. If static priorities are used, Throughout the complete execution of the program, the priority of each task remains constant, while where dynamic priorities are used, the priority of the task can change at any time.

**Rate monotonic scheduling** is a priority-based scheduling algorithm used in real-time embedded systems with static(fixed) priorities which is assigned at design time and remains the same till the end of the process. It is designed to make better use of hardware resources to meet real-time requirements.

## 3.2 Rate Monotonic Scheduling

Introduced first by Liu and Layland with sufficient tests, they managed to introduce the concept of achievable utilization factor through set of pre-conditioned tests (though pessimistic) to derive a low-complexity check that is used to evaluate the schedulability of discrete, periodic, and preemptive task sets performed on one processor.

the Rate Monotonic Scheduling (RMS) algorithm could indeed guarantee schedulability. The rate monotonic scheduling algorithm simply states that the higher its priority should be, the more frequently a task runs (the higher its frequency). Clearly, the most frequent task has the highest priority in the system. If an operation is of course completely periodic or can be done so, Then the developer is lucky because the RMS algorithm can be used, and the application can be provably correctly asserted to meet its deadlines.

It is based on several assumptions:

1. All tasks are periodic with constant intervals between requests. (occur at regular intervals)
2. Deadlines are reasonable. (every task must be done before it’s next request)
3. Tasks must be independent and do not synchronize with another or share resources.
4. Preemptive scheduling must be used. (higher priority task should run first)
5. Static priorities and the task with the higher priority immediately preempts any other task.
6. Context switching time is ignored.

Rate monotonic scheduling looks at a run simulation of all threads in the system and calculates how much time it takes to fulfill the thread set guarantees. Its priority assignment is also optimal which means that if there is any other static priority-based scheduling algorithm’s running tasks can be completed before the deadlines then the Rate monotonic scheduling algorithm can meet the deadlines too.

It’s compared a lot with Earliest deadline first(EDF) algorithm(it was also introduced by Liu and Layland) that has more efficient in the use of the computational resources than Rate monotonic scheduling but still, it uses dynamic priorities instead of fixed priorities. However, rate monotonic scheduling has a few advantages:

1. Lower computational overhead.
2. Simple to implement.
3. Predictable.

It’s also commonly used on most real-time operating systems and is endorsed by most standards of real-time systems.

**Example 1 on Rate-Monotonic Scheduling**

|  |  |  |
| --- | --- | --- |
| Process | Execution time | Period |
| P1 | 1 | 4 |
| P2 | 2 | 6 |
| P3 | 3 | 12 |

we have a set of processes, their execution time, and their period. // table

using rate monotonic scheduling we give p1 high priority, p2 medium priority, p3 lowest priority. And to get a better understanding of the interaction between these tasks, we build a timeline with the length of the least common multiple of all the processes periods which is 12 in this example. When we use the least common multiple of the periods the scheduler is called in this case the unrolled schedule.

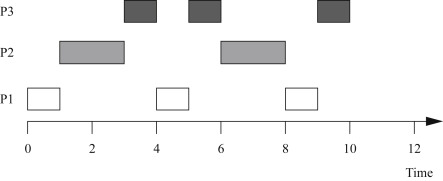


figure - RMS Example 1

All three periods start at time zero. P1's data arrive first. Since P1 has the highest priority it starts executing first, after a time unit, P1 finished execution and block for the rest of its period, at this time P2 starts it’s execution as it’s currently the highest priority processes, at time 3, P2 finished and blocks for its remaining period and P3 starts executing, at time 4, P1 exits it’s blocked state and interrupts P3, after P1 finishes P3 executes for 1 time unit and gets interrupted by P2, then P1 exits it’s blocked state and executes again as the highest priority process, then finally P3 gets its chance to finish execution at time 9 after the third iteration of P1.

If we now change the execution time for the processes but keep the same deadlines.

|  |  |  |
| --- | --- | --- |
| Process | Execution time | Period |
| P1 | 2 | 4 |
| P2 | 3 | 6 |
| P3 | 3 | 12 |

In this case, we can show that there is no feasible assignment of priorities that guarantees scheduling. Although each process has execution time less than its period combing them might require more than 100% CPU utilization. For example, during one 12 time-unit interval, we must execute P1 three times, requiring 6 units of CPU time; P2 twice, costing 6 units of CPU time; and P3 one time, requiring 3 units of CPU time. The total of 6 + 6 + 3 = 15 units of CPU time is more than the 12 time units available, clearly exceeding the available CPU capacity.

## **3.3 Utilization**

The bad news is that while Rate Monotonic Scheduling is the ideal static priority schedule, the algorithm can not use 100% of the available CPU cycles.

Based on Liu and Layland tests in 1973 they derived a least utilization upper bound under a set of conditions that tells us whether the set of tasks are guaranteed schedulability or may not be schedulable.

* n for tasks, T for task period, C for task execution time, U for total utilization.
* For n tasks, τ1, τ2, … τn the utilization factor U = C1/T1 + C2/T2 + … + Cn/Tn
* U for a task set Γ can be improved by increasing Ci’s (slower processor) or
* decreasing Ti’s as long as tasks continue to meet deadlines at critical instants
* U=1 means that the processor is fully utilized

It can be seen that the CPU utilization U has a minimum upper bound of 2(2^(1/2) − 1) ≅ 0.83.for a set of two tasks under RMS scheduling. In other words, at least 17 percent of the time the CPU will be idle. This idle time is because priorities are automatically assigned; in the next section we see that more proactive scheduling practices will boost the usage of CPUs.

As n approaches infinity, the utilization of the CPU (with the factor-of-two constraint on the relationship between periods) approaches asymptotically ln 2 = 0.69—the CPU will be idle 31 percent of the time. We can use processor use U as an easy measure of the feasibility of an Rate Monotonic Scheduling issue.

For example, if we have a Rate Monotonic schedule system which P1 has a period of 4, execution time of 2, P2 has period of 7 and execution time of 1, in this case we satisfy the factor-of-two restriction on each period. The cycle hyperperiod is 28, so the CPU utilization of this group of processes is [(2 × 7) + (1 × 4)]/28 = 0.64, which is less than our ln 2 limit.

## **3.4 Conclusion**

The Rate Monotonic Scheduling is the optimal algorithm for fixed-priority preemptive RTOS.

If the total utilization is less than n(2^(1/n)-1) then all tasks will go according to the scheduling and finish before the time limit.

If the total utilization is greater than n(2^(1/n)-1) then tasks may or may not be schedulable and need further analysis of this task set,

We can achieve 100% utilization with fixed priorities, if the periods of tasks are harmonic, this means each task has a period that is a multiple of tasks with shorter periods.

For example, three tasks with periods, 10, 20, 40 respectively are considered harmonic and are preferred over the task with 10, 20, 50 periods.

# References

Arnoldo Díaz-Ramírez, Pedro Mejía-Alvarez, Luis E. Leyva-del-Foyo. "Comprehensive Comparison of Schedulability Tests for Uniprocessor Rate-Monotonic Scheduling." *Journal of Applied Research and Technology* 11, no. 3 (2013): 408-436.

Liu, Chung Laung, and James W. Layland. "Scheduling algorithms for multiprogramming in a hard-real-time environment." *Journal of the ACM (JACM)* 20, no. 1 (1973): 46-61.

Stewart, Dave. “Introduction to Real Time,” *Embedded Systems Programming.* 2001.

Walls, Colin. “Real-Time Operating Systems” *Embedded software: the works.* Elsevier, 2012.

Wolf, Marilyn. “Process and Operating Systems” *Computers as Components.* 4th ed. 2017.