ECE 4550/5550G Final Project

Implementation of Polling Server in FreeRTOS

Richa Singh
richas@vt.edu
Virginia Tech

FNU Sachin
sachin255701@vt.edu
Virginia Tech

Table of Contents

Introduction

In this project, we present the implementation of Rate Monotonic Scheduling algorithm with integrated Polling Server to service aperiodic job requests using open source FreeRTOS platform. This report will present the details of the approach for implementation of Rate Monotonic Scheduling algorithm along with integrated Polling Server. Project report also encloses the complete analysis of both algorithms from performance standpoint. For the implementation and analysis part of the project, Arduino Mega 2560 hardware is used to run, debug and verify the algorithms.

Rate Monotonic Scheduling Algorithm

Rate Monotonic is a static priority algorithm that assigns the priorities to the periodic tasks according to their rate of occurrence. The lower the rate of occurrence i.e. shorter the period, higher is the priority assigned. As periods of periodic tasks are constant, that is why it is called fixed/static priority scheduling algorithm. Each periodic task has some parameters that characterizes that task. These parameters are its priority, deadline, period and worst-case execution time.

Polling Server

Rate Monotonic Scheduling algorithm only deals with the periodic tasks. However, in real-time control applications, may consist the presence of both periodic and aperiodic tasks simultaneously. Such type of task set is known as hybrid task set and consists of both periodic and aperiodic jobs. Aperiodic jobs are the one that can occur at any random time interval during the normal execution of periodic tasks, so their arrival time is not known to the system beforehand. So, concept of Polling Sever is developed to serve both periodic and aperiodic task in hybrid task sets.

Polling server is also a periodic task, whose job is to service the pending aperiodic requests. Like each and every periodic task, the Polling Server task has priority, deadline, time period and server capacity (worst-case execution time). Polling Server, when active, executes any pending aperiodic task within its capacity. If no aperiodic task is pending then the polling server will suspend itself until the next period and at every periodic instant the capacity of the server is refreshed. Figure 1 shows the timeline of two periodic tasks and a polling server with highest priority. This timeline nicely captures the essence of working polling server as explained above.

Rate Monotonic Scheduling Using FreeRTOS

Rate Monotonic scheduling algorithm is implemented using FreeRTOS in $scheduler.cpp$ and $scheduler.h$ files. Rate Monotonic scheduling algorithm works by scheduling the task set that has shortest period with highest priority and then other task in the task set also follows the same criteria based on their time periods. Abstraction layer on top of FreeRTOS code is implemented to realize RM scheduling algorithm for periodic tasks.

RM implementation uses extended Task Control Block $SchedTCB\_t$, which is a structure that provides various fields for managing and tracking all periodic tasks in task set. Task array $xTCBArray$ of type $SchedTCB\_t$ is used to keep track of task control blocks of each task.

Apart from the tasks that are provided in the given task set, we create another task called Scheduler Task, that is responsible for checking if any task has missed its deadline or has exceeded its worst-case execution time limit. This task is assigned the highest priority, equal to $schedSCHEDULER\_PRIORITY$. $prvSchedulerFunction$ contains the code for this scheduler task. In this function $prvSchedulerCheckTimingError$ is being called for each task in task array.

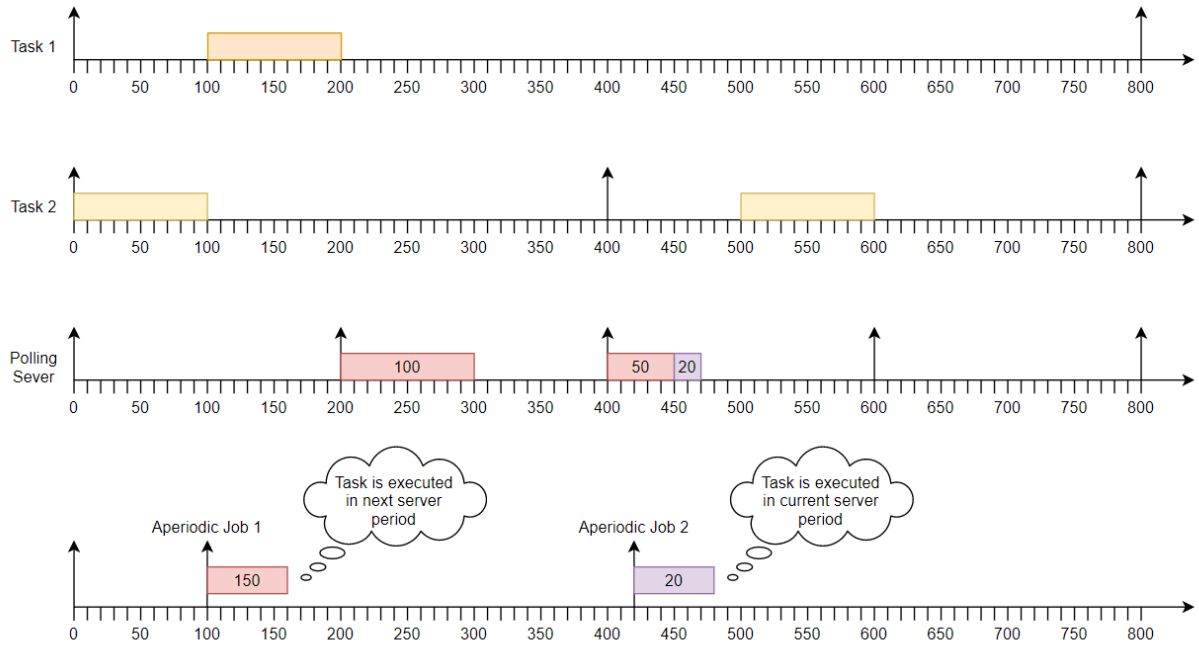

Figure 1. Worst-case scenario for 3 periodic tasks and a Polling Server (PS) with the highest priority

Following functions, in $scheduler.cpp$ file, are used to implement RM scheduling algorithm:

$\boldsymbol{vSchedulerPeriodicTaskCreate}$ function is used to populate the extended task control block $SchedTCB\_t$ entries as per the specifications given in the task set. This field populates entries which corresponds to specifications like task priority, task period, task handle, task deadline, task maximum execution time etc. Apart from these, task specific information, this function also helps in keeping the count of total number of tasks by incrementing $xTaskCounter$ every time it is called. This function also helps in initializing various task control block entries like $xSuspended$, $xMaxExecTimeExceeded$ and $xExecutedOnce$, which are being used while checking if a task has exceeded its worst case execution time or has exceeded its deadline.

$\boldsymbol{prvSetFixedPriorities}$ function is used to assign priorities to the task as per Rate Monotonic scheduling algorithm. In this function, all tasks entries in $xTCBArray$ are parsed to assign the priorities in to these tasks in ascending order of their time period values, giving highest priority to the task with shortest $xPeriod$. This function is capable to handle tasks that have same $xPeriod$ values and assign those tasks same priority value.

prvCreateSchedulerTask function creates scheduler task with highest priority $schedSCHEDULER\_PRIORITY$. $prvSchedulerFunction$ function is executed every time scheduler task is invoked.

prvCreateAllTasks function creates task described in the task set using FreeRTOS API $xTaskCreate$. This function uses information, like $pcName$, $uxStackDepth$, $pvParameters$, $uxPriority$ and $pxTaskHandle$, from $xTCBArray$ to create each task. For each task, task function is set to $prvPeriodicTaskCode$. So, every time a task is invoked, $prvPeriodicTaskCode$ function is called, which manages execution of each task.

prvSchedulerFunction function is called every time when the scheduler task is invoked. Whenever this function is called its responsibility is to check whether any task as exceeded its deadline. This function calls $prvSchedulerCheckTimingError$ to perform this functionality. Once done with the deadline error checking, the scheduler task is blocked indefinitely using API $ulTaskNotifyTake$. This API blocks this task till it is being notified using API $vTaskNotifyGiveFromISR$.

prvWakeScheduler function is used to notify the blocked scheduler task and resume the execution of that task. To resume the execution of the task, API $xTaskResumeFromISR$ is used which resumes a suspended task and can be called from within an ISR.

vApplicationTickHook function is called at every software tick. In this function, each task (other than scheduler task) check whether its has not exceeded its worst-case execution time. For this purpose, $xExecTime$ entry of extended task control block of the task is updated every time this function is called. Whenever $xExecTime$ value exceeds $xMaxExecTime$ for a particular task, this implies that task has executed more than its worst case execution time, and calls function $prvExecTimeExceedHook$ to take necessary actions such as preparing for suspending the task. The function $vApplicationTickHook$ is also responsible for waking up blocked scheduler task. Every time when this function is called the value of $xSchedulerWakeCounter$ increases every time and when count reaches $schedSCHEDULER\_TASK\_PERIOD$, then scheduler is woken up calling $prvWakeScheduler$. $schedSCHEDULER\_TASK\_PERIOD$ has value equal to number of ticks equivalent to $100\ ms$. So, scheduler task is notified every $100\ ms$ from this function.

prvSchedulerCheckTimingError this function is called by $prvSchedulerFunction$ function to check whether any task has missed its deadline. This function calls $prvCheckDeadline$ function to check if has task has exceeded its deadline or not. This function also helps in suspending or resuming the task in case the task has executed more than its worst-case execution time. If the $xMaxExecTimeExceeded$ entry of task control block of the task is set to $pdTRUE$, this indicates that the task has executed more than its worst-case execution time and must be suspended using API $vTaskSuspend$. However, if $xSuspended$ entry of task control block of the task is equal to $pdTRUE$, then decision to resume the task is made based on current tick count and $xAbsoluteUnblockTime$, if current tick count is greater than or equal to $xAbsoluteUnblockTime$ then the suspended task is resumed, else not.

prvExecTimeExceedHook function is called from function $vApplicationTickHook$. It is called if any task has executed more that its worst-case execution time. In this function, $xAbsoluteUnblockTime$, which is the time after which the suspended task should be resumed, is set to sum of $xLastWakeTime$ and $xPeriod$. After that, the task is blocked and the scheduler task is invoked to suspend the task.

prvCheckDeadline function is used to check if the task has exceeded its deadline. $xAbsoluteDeadline$ entry of the extended task block of the task is updated. It is checked if current tick count is greater than

$xAbsoluteDeadline$ of the task, then the task has missed its deadline and should be deleted. $prvDeadlineMissedHook$ function called to delete and recreate the task.

$prvDeadlineMissedHook$ function is used to delete and recreate the task that has missed its deadline. For deleting the task API $vTaskDelete$ is used which removes a task from all list be it ready, blocked or suspended list. Then the deleted task is recreated using $prvPeriodicTaskRecreate$ function.

$prvPeriodicTaskRecreate$ function is used to re-create the task that has been deleted because it missed its deadline. If the task creation is successful, then three task parameters, $xExecutedOnce$, $xSuspended$ and $xMaxExecTimeExceeded$ is set to $pdFALSE$. $xExecutedOnce$ is set to false to make sure that the re-created task doesn't immediately miss its deadline, rest others are set as new task should have these task code block entries set to $pdFALSE$.

Polling Server Using FreeRTOS

Polling Server is a periodic task, which is responsible to service pending aperiodic requests. Being a periodic task, the Polling Server has various parameters such as priority, maximum execution time, task deadline, period of the task etc. These parameters are defined in $scheduler.h$ in the form of following four macros:

1. $POLLING\_SERVER\_PRIORITY$
2. $POLLING\_SERVER\_PERIOD$
3. $POLLING\_SERVER\_MAX\_EXEC\_TIME$
4. $POLLING\_SERVER\_RELATIVE\_DEADLINE$

Just as we created scheduler task using the function $prvCreateSchedulerTask$, the Polling Server task is created using function $prvCreatePollingServerTask$. This function uses above given macros related to the Polling Server task, to create a task with desired parameters. $pollingSeverTaskHandle$ is used as the task handle for Polling Server task.

Aperiodic jobs are implemented in the form of functions that must be executed by the Polling Server. Aperiodic job parameters are managed and tracked using a structure $AJTCB\_t$. Following are the members of this structure:

1. $pvTaskCode$: This element holds the pointer to the function that must be executed by the Polling Server to service the aperiodic request.
2. $pcName$: This element specifies the name of the aperiodic task.
3. $pvParameters$: This element of the structure holds any parameters that must be passed to the task function during the execution.
4. $pxTaskHandle$: This element holds the task handle that is used to identify the task.

Incoming pending aperiodic requests are managed using a global FIFO queue, namely $aperiodicTCBQueue$. FIFO queue, shown in Figure 2, is implemented using an array with elements of type $AJTCB\_t$ and two pointers namely $queueHead$ and $queueTail$. Array size is defined equal to the number of permitted aperiodic jobs in the system, which is controlled by macro defined in $scheduler.h$. The name of the macro is $schedMAX\_NUMBER\_OF\_APERIODIC\_TASKS$, and has a default value of 3. Two pointers are used to track the pending aperiodic requests. Pointer $queueHead$ points to the front of queue and points to the aperiodic job which must be extracted from the queue. Pointer $queueTail$ points to next available location in array where a new incoming request can be enqueued. A separate counter, $aperiodicJobCounter$, is also

maintained to count the total number of pending jobs in the queue. Following is the visual structure of the queue mechanism used for managing aperiodic jobs:

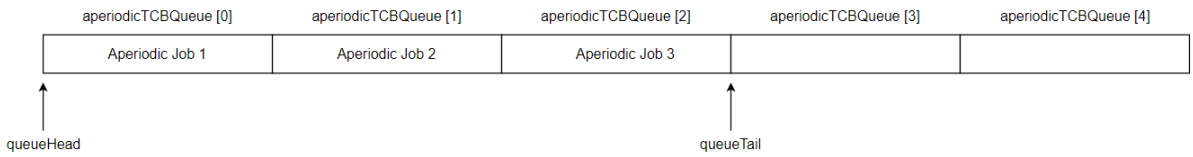

Figure 2. Design of $aperiodicTCBQueue$

Every time the Polling Server task starts its execution, it retrieves aperiodic job parameters from the $queueHead$ pointer. Once an aperiodic job is fetched from the queue, the $queueHead$ pointer is incremented such that it points to next element of the queue. Similarly, when any aperiodic job is inserted in the queue, the $queueTail$ pointer is incremented. This queue is a circular queue which is capable of holding $schedMAX\_NUMBER\_OF\_APERIODIC\_TASKS$ elements at a time.

Following functions are used to create and execute aperiodic jobs:

createAperiodicJob: This function is used to create aperiodic job and enqueue that job into the FIFO queue. This function calls function $getEmptyIndexInQueue$ to get the index of empty location inside the queue. If empty location is found inside the queue then the parameters of the structure $AJTCB\_t$ are updated as per aperiodic request and status $pdTRUE$ is returned. If no empty location is found in the queue, the function will return $pdFALSE$ status. $aperiodicJobCounter$ is incremented every time a job is created.

executeAperiodicJob: This function is the task code for Polling Server task. This function is used to execute an aperiodic job that is pending in the FIFO queue. If no aperiodic job is pending, then nothing is done and the execution is returned. If an aperiodic job is pending in the queue, then a job is fetched from queue then the function pointed by $pvTaskCode$ is executed. This function will keep fetching aperiodic jobs from queue until the queue is empty or the Polling Server task has exceeded its worst-case execution time. If an aperiodic job is executing and the Polling Server task has exceeded its worst-case execution time, then the aperiodic job execution will be resumed in the next task period. Every time a job is executed, $aperiodicJobCounter$ is decremented.

getEmptyIndexInQueue: This function is used to find the index of empty location in queue. This function returns -1 when no empty location is there in the queue otherwise returns the index of the empty location in the queue. If an empty location is found then the tail pointer is incremented to point to next empty location in the queue.

Performance Analysis

In this project, the extensive performance analysis of polling server for aperiodic jobs in terms of their average response times is performed by varying the design parameters of the Polling Server task such as Server Period and Server Capacity. For this project, we have used opensource FreeRTOS real-time operating system kernel. The evaluation of the implementation is done using Arduino Mega R3 2560 board and Arduino Uno board.

Server Period vs Server Capacity Impact Analysis: This experiment is conducted with two periodic tasks, polling server and variable number of aperiodic job instances. Periodic task functions are adapted to create aperiodic jobs at configured time periods. Aperiodic jobs generated through this method fall in the worst-case scenario, that is, aperiodic jobs will have to wait for at least one server period before being serviced by polling server. This method helped to set the environment for the worst-case response time analysis of aperiodic jobs. Polling Server is created with an execution time of 100ms and time period of 200ms using Rate Monotonic algorithm. Table 1 defines the task parameters for periodic tasks taken for this experiment.

Periodic Task Parameters			
Task Name	**WCET (ms)**	**Time Period (ms)**	**Deadline (ms)**
Polling Server	100	200	200
Task 1	100	800	800
Task 2	100	400	400

Table 1. Periodic Task Parameters

Case 1: Server period is varied and Server capacity is kept constant for a fixed set of two aperiodic jobs with 3 instances each. It is observed that as the server period increases, average response times of aperiodic jobs 1 and 2 also increases. Table 2 shows the data captured using a task set as per specifications mentioned above. Observations captured in the table are as per expectation because longer server period implies polling server being assigned as the low priority task hence polling server gets the last time slice to service aperiodic jobs which have arrived earlier. Hence, this increases the response time of the aperiodic jobs.

Response Time when Server Capacity = 100 ms		
Server Period (ms)	**Aperiodic Task 1 (ms)**	**Aperiodic Task 2 (ms)**
200	376	447
	306	440
	372	474
Average	**351.3333333**	**453.6666667**
400	679	817
	478	615
	503	780
Average	**553.3333333**	**737.3333333**
800	1728	3528
	1493	2354
	1153	1412
Average	**1458**	**2431.333333**

Table 2. Response Time when Server Period = 400ms

Case 2: Server capacity is varied and Server period is kept constant for a fixed set of two aperiodic jobs with 3 instances each. Aperiodic jobs are configured such that their execution time is greater than the server capacity.

It is observed that for a fixed server period, as the server capacity decreases, average response times of aperiodic jobs 1 and 2 also increases. Table 3, Table 4 and Table 5 shows the data captured using a task set when Server Period is kept equal to 200ms, 400ms and 800ms respectively. Observations captured in the table are as per expectation because aperiodic job gets less executed in the current server period due to decreased server capacity hence its execution overflows to next server period and can take subsequent server periods for complete service.

Response Time when Server Period = 200 ms		
Server Capacity	**Aperiodic Task 1 (ms)**	**Aperiodic Task 2 (ms)**
50 ms	541	651
	475	643
	512	645
Average	**509.3333333**	**646.3333333**
100 ms	376	447
	306	440
	372	474
Average	**351.3333333**	**453.6666667**

Table 3. Response Time when Server Period = 200ms

Response Time when Server Period = 400 ms		
Server Capacity	**Aperiodic Task 1 (ms)**	**Aperiodic Task 2 (ms)**
50 ms	1639	1398
	1506	2236
	1702	1604
Average	**1615.666667**	**1746**
100 ms	679	817
	478	615
	503	780
Average	**553.3333333**	**737.3333333**

Table 4. Response Time when Server Period = 400ms

Response Time when Server Period = 800 ms		
Server Capacity	**Aperiodic Task 1 (ms)**	**Aperiodic Task 2 (ms)**
50 ms	2582	6366
	11047	7405
	51699	51619
Average	**21776**	**21796.66667**
100 ms	1728	3528
	1493	2354
	1153	1412
Average	**1458**	**2431.333333**

Table 5. Response Time when Server Period = 800ms

Case 3: Server capacity and Server period is kept constant and execution time of aperiodic job with 5 instances is varied. In this case, Polling server is assigned the highest priority. It is observed that as the execution time of the aperiodic job instances is increased, their average response time also increases when server parameters are constant. The observations are captured in Table 6. When mean execution time of 5 instances of aperiodic job is 68.2 ms, each instance gets completed within two server periods. As the mean execution time of aperiodic job instances is increased, their average response time also increases showing that for the same server capacity and server period, it takes a greater number of server periods to get completely executed.

Average Response Time vs Mean Execution Time	
Execution Time (ms)	**Response Time (ms)**
68.2	136
274.6	336.2
447.2	501.4
692.4	794.8
877.8	1001.4
1134.8	1251

Table 6. Average Response Time vs Mean Execution Time

We further experimented the tradeoff between aperiodic load and average response time where aperiodic load represents the number of aperiodic jobs with arbitrary arrival time. Figure 3 shows that as the aperiodic load in the system increases, their average response time also increases. Here, periodic tasks including polling server has the task parameters as depicted in Table 1.

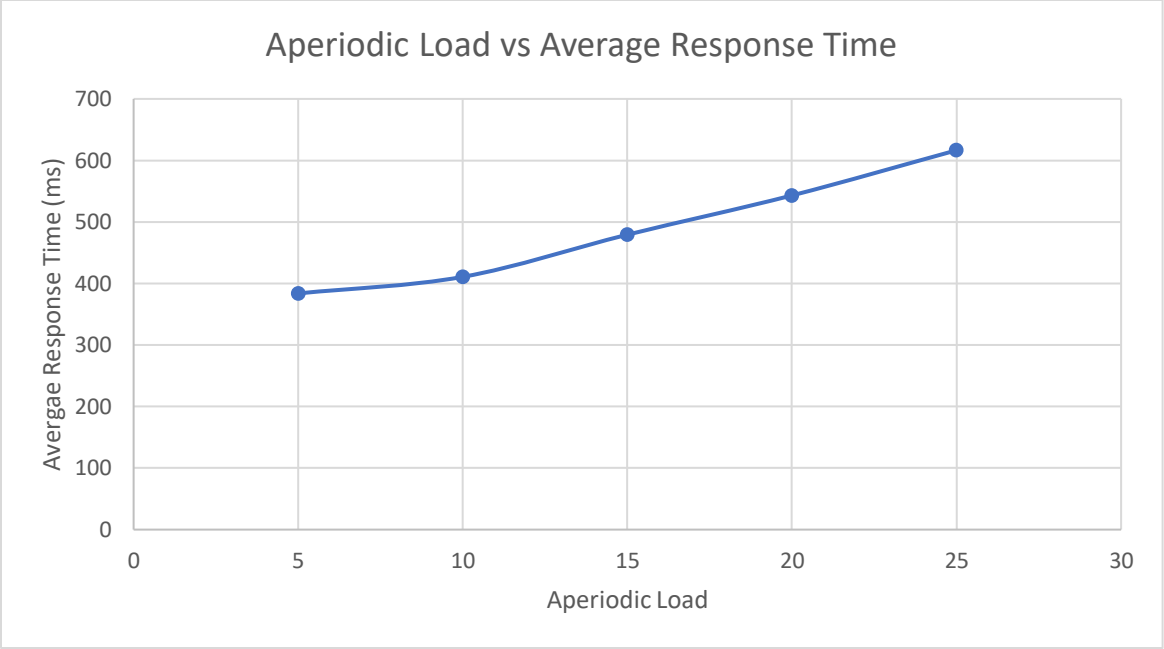

Figure 3. Tradeoff between aperiodic load and average response time

This project has also verified the correctness of implementation of polling server by demonstrating a scenario where polling server is not able to immediately service aperiodic jobs. This leads to longer aperiodic response time and is the major problem of polling server. Below is the snippet of the serial monitor which captures this scenario.

The serial monitor output shows that polling server executes first, followed by periodic task 2 and then periodic task 1 in the order of their decreasing priority. When no aperiodic job has arrived, polling server checks for any aperiodic job in the queue, it finds the queue empty then suspends itself and gets activated at next server time period as shown in the snippet. It is clear from the snippet that when aperiodic job apt1 arrives, polling server has already suspended itself as a result it does not get serviced immediately and waits until the beginning of next polling server period. In next polling server period, apt1 starts execution and executes until polling server capacity is fully consumed.

Conclusion

From above analysis, we conclude that Polling Server has a major drawback when scheduling aperiodic tasks with a soft deadline. Polling server is not able to service aperiodic tasks immediately and this makes polling server unsuitable for minimizing the aperiodic response time. We explored the design space of polling server by varying its parameters and analyzed its impact on average aperiodic response time.