|  |  |  |  |
| --- | --- | --- | --- |
| BN | Section | Student ID | اسم الطالب |
| **42** | **3** | **9210950** | **محمد جاد محمد احمد** |
| **17** | **3** | **9210808** | **فتحي مصطفي فتحي عبد الحميد** |

# System Design

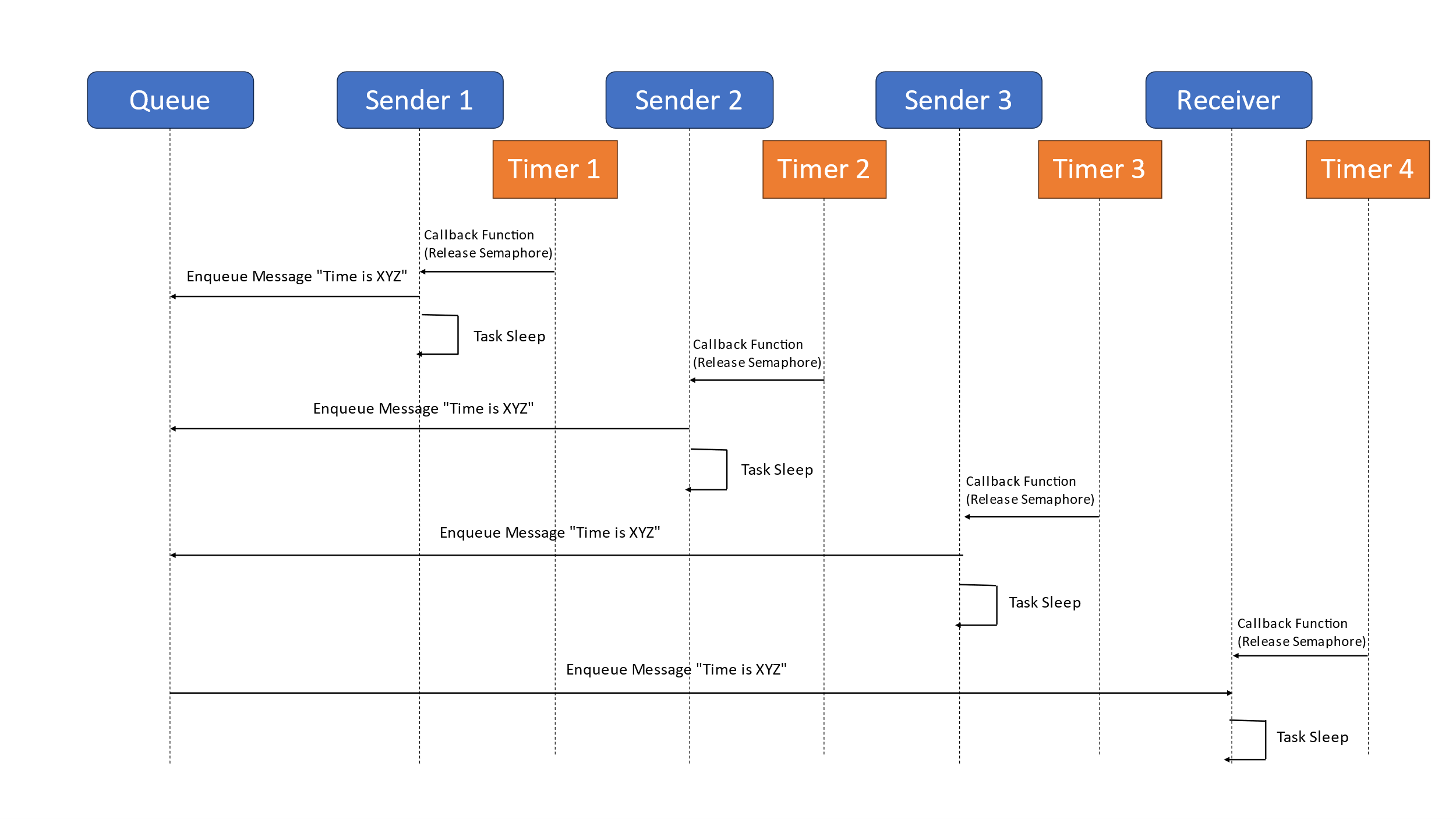
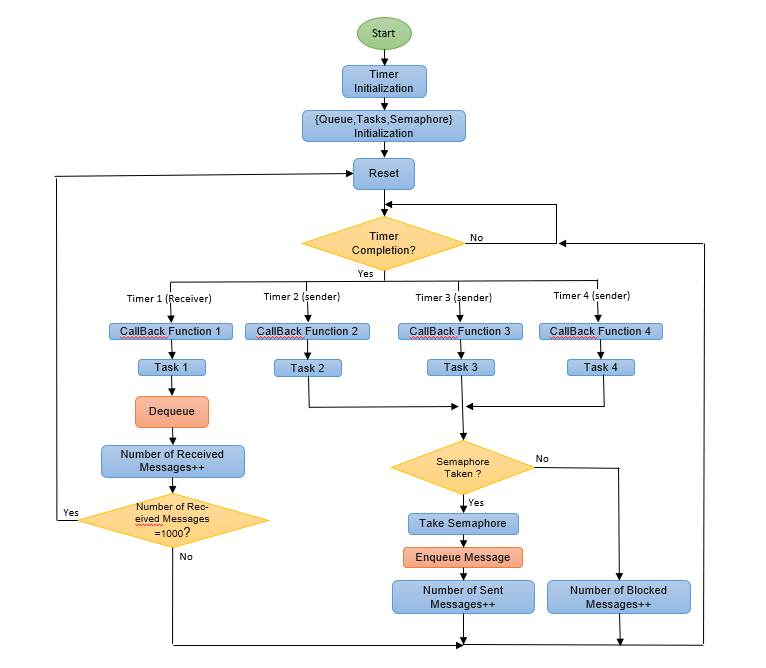


Figure 1:Message Sequence Figure 2:Flow Chart of the System

There is a priority to execute in the task, for example, task 1 has the highest priority because it is the receiver, and tasks 1 and 2 have the lowest priority because task 3 needs a strict deadline for execution, and if one of the tasks is in the execution state and another task is called, then we check Priority, and if the priority is greater, we cut off the first task and store its registers and execute the second task, and when it is finished, we return to the task with the lowest priority, and this priority cannot be changed unless the task with the highest priority needs a semaphore taken in the task of the lowest priority, then we continue Task is the lowest priority until it releases the semaphore.

# Results and Discussion

To represent the system data, we printed the process status statement from Success or Block, then we made a text processing using the MATLAB, the following table shows the output.

Comment:: We put our MATLAB code, variables and console outputs in [GitHub Repository](https://github.com/faatthy/rtos-project) .

Table 1: Overview of the Results and Statistics

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Iteration** | **Queue** | **1** | **2** | **3** | **4** | **5** | **6** |
| **Average Of Sent Messages Time for Sender 1** | **3** | **98.3** | **139.976** | **179.973** | **221.144** | **261.717** | **302.926** |
| **10** | **98.666** | **139.473** | **181.751** | **219.331** | **259.567** | **301.832** |
| **Average Of Sent Messages Time for Sender 2** | **3** | **98.96** | **138.978** | **180.5** | **220.24** | **260.161** | **302** |
| **10** | **98.586** | **140.351** | **180.924** | **220.881** | **260.628** | **302.423** |
| **Average Of Sent Messages Time for Sender 3** | **3** | **98.85** | **140.12** | **182.659** | **222.463** | **257.918** | **300.929** |
| **10** | **98.858** | **139.37** | **181.072** | **224.84** | **260.916** | **302.724** |
| **The total number of Successfully sent Messages** | **3** | **1002** | **1002** | **1002** | **1001** | **1002** | **1002** |
| **10** | **1009** | **1009** | **1009** | **1009** | **1009** | **1001** |
| **the total number of Blocked Messages** | **3** | **2035** | **1145** | **655** | **353** | **152** | **13** |
| **10** | **2028** | **1139** | **648** | **346** | **145** | **0** |
| **Blocked Messages for Sender 1** | **3** | **640** | **357** | **219** | **113** | **52** | **4** |
| **10** | **686** | **385** | **207** | **112** | **50** | **0** |
| **Sent Messages Time for Sender 1** | **3** | **376** | **358** | **337** | **339** | **330** | **333** |
| **10** | **327** | **332** | **343** | **344** | **336** | **334** |
| **Blocked Messages for Sender 2** | **3** | **712** | **395** | **211** | **10** | **50** | **3** |
| **10** | **653** | **379** | **242** | **116** | **50** | **0** |
| **Sent Messages Time for Sender 2** | **3** | **298** | **324** | **343** | **324** | **335** | **335** |
| **10** | **360** | **334** | **311** | **337** | **334** | **333** |
| **Blocked Messages for Sender 3** | **3** | **683** | **323** | **225** | **110** | **50** | **6** |
| **10** | **689** | **375** | **199** | **118** | **45** | **0** |
| **Sent Messages Time for Sender 3** | **3** | **328** | **320** | **323** | **339** | **338** | **334** |
| **10** | **322** | **342** | **354** | **327** | **338** | **333** |

Average of Sent Messages Time for Sender Task is Almost equal the timer lower and upper bounds which mean that the function which generate the random values is uniformly distributive.

The total number of Successfully sent Messages equal the received messages plus the size of the queue if the rate of message sending is greater than the rate of receiving.

**Explaintion of the gap between the number of sent and received messages:**

The total number of sent Messages equal the Successfully sent Messages plus the Blocked Messages which mean that the gap between the number of sent and received messages in the running period is the message which still inside the queue after 1000 message received and the Blocked messages which could not sent Successfully because the queue is full.

The gap decreases as the sender timer period increases because it allows the sender task to send messages at a slower rate, reducing the likelihood of the queue becoming full. Eventually, when the receiver timer has a smaller period than the sender timer, the gap reaches zero. In this scenario, the receiver task is receiving messages at a faster rate than the sender task is sending them, ensuring that the queue doesn't become full and eliminating any gap between sent and received messages.

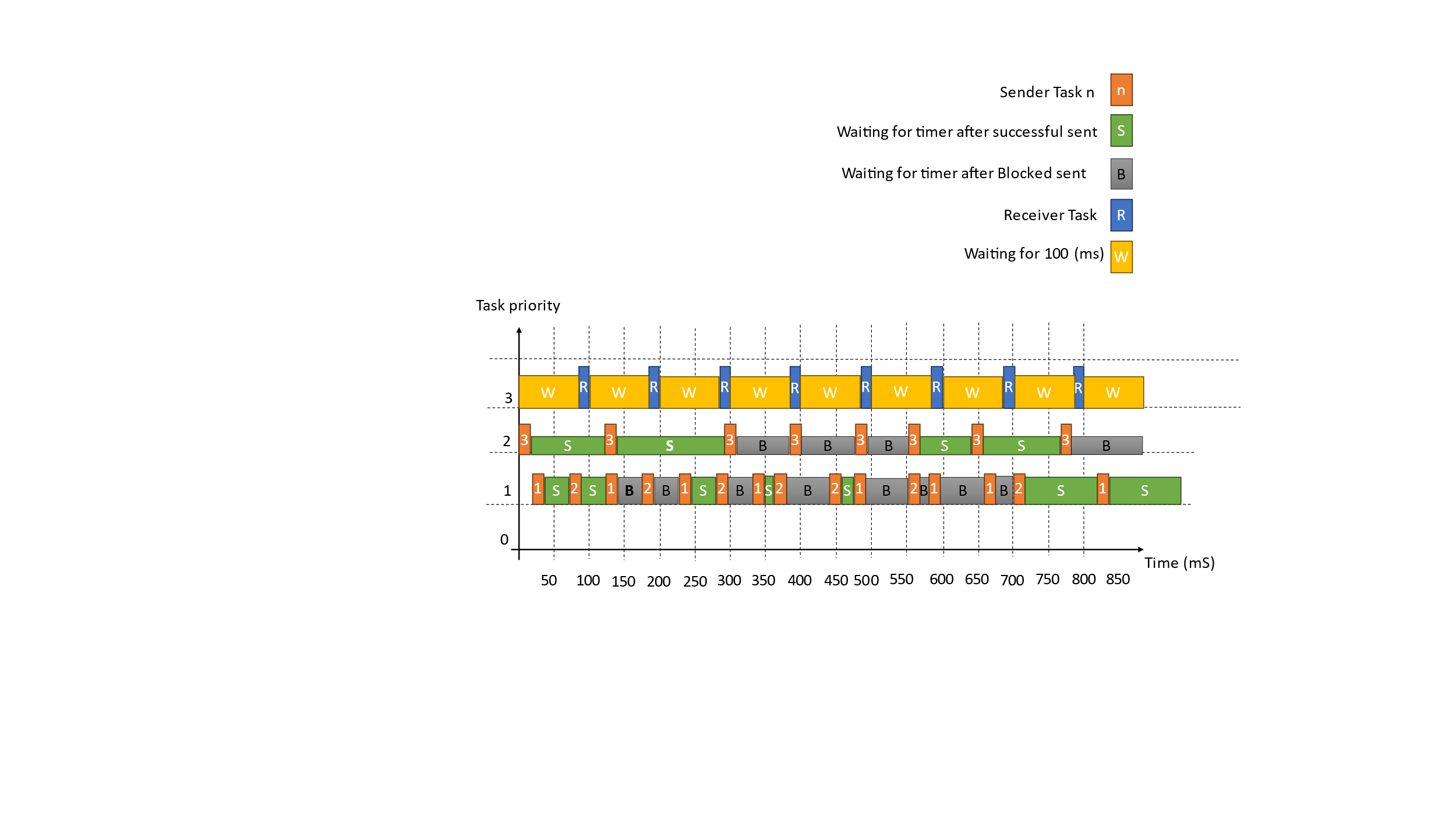


Figure 3: Tasks Flow with the priority (Queue size=3)

This Figure show the first 8 Iteration of our system with queue size = 3 , and from the figure We can conclude that when the queue is full, the first Task will execute after the Receiver Task will send the message successfully and the other Task’s messages will be blocked until the next Iteration.

رمز "تم التحقق منها بواسطة المنتدى"

|  |  |
| --- | --- |
| **Queue size = 3** | **Queue size = 10** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Increasing the size of the queue can reduce the number of blocked messages and increase the number of successfully sent messages. With a larger queue, it takes more time for messages to accumulate and fill up the available space. This means that during this extended period, messages are not blocked because there are still empty spaces in the queue to accommodate them.

By allowing more room for messages in the queue, the system can handle a higher volume of messages without encountering blocking conditions. This results in a higher number of successfully sent messages, as fewer messages are prevented from being transmitted due to a full queue. Therefore, increasing the queue size can help mitigate the issue of blocked messages and improve overall message transmission efficiency.

Comment: Our GitHub repository link:

https://github.com/faatthy/rtos-project

## Code Snippets

/\* Uniform Distribution (Function one)\*/

int uniform\_distribution(int Range\_Low, int Range\_High)

{ int Random\_Value = (int)rand();

int Range = Range\_High - Range\_Low + 1; //+1 makes it [rangeLow, rangeHigh], inclusive.

int Scale = (Random\_Value % Range) + Range\_Low;

return Scale;}

/\* Uniform Distribution (Function two)\*/

int uniform\_distribution(int Range\_Low, int Range\_High)

{ srand(time(NULL));

int Random\_Value = rand();

int Range = Range\_High - Range\_Low ;

int DEV = RAND\_MAX / Range;

Random\_Value = Random\_Value / DEV;

int Scale = Random\_Value + Range\_Low;

return Scale;}

**Uniform Distribution function**

To get Uniform Distribution Values we can use two algorithms, first function we used the modulus to get the random values, and the second one we used time seeding to make it more random and normalize it.

In our code we used the first function as the second one wasn’t totally Uniform Because of the time seeding.

**Sender Task**

void Sender\_Task\_1(void \*p) //Sender 2 Task

{

TickType\_t Current\_Time; //Variable to store current time

char Data\_Sent[20]; //Variable to store data that will be sent to Queue

while(1){

if(xSemaphoreTake(Sender\_1, portMAX\_DELAY)){

Current\_Time = xTaskGetTickCount();

sprintf(Data\_Sent, "Time is %d", Current\_Time);

if(xQueueSend(Queue,Data\_Sent,0)){

Number\_of\_Sent\_Messages++;

puts("Sender 1");

}

else{

puts("Sender 1 Blocked");

Number\_of\_Blocked\_Messages++;

} } } }

Sender\_Task\_1 and Sender\_Task\_2 have the same priority, while Sender\_Task\_3 has a higher priority, the scheduling behavior of the tasks would be affected. In an operating system with priority-based scheduling, tasks with higher priorities are given preference over tasks with lower priorities when it comes to CPU time allocation.

In this scenario, when all three sender tasks are ready to run, the task scheduler would prioritize Sender\_Task\_3 over Sender\_Task\_1 and Sender\_Task\_2 due to its higher priority. This means that Sender\_Task\_3 would be executed first, allowing it to acquire the semaphore (Sender\_3) and attempt to send a message to the queue. If the queue is not full, the message would be sent successfully, and the appropriate counters would be incremented.

Once Sender\_Task\_3 completes its execution or gets pre-empted by a higher-priority task, the task scheduler would consider the remaining sender tasks. Since Sender\_Task\_1 and Sender\_Task\_2 have the same priority, the scheduler would likely employ a round-robin scheduling approach, allowing each task to execute for a specified time quantum or until it blocks or completes its execution.

The round-robin scheduling would alternate the execution between Sender\_Task\_1 and Sender\_Task\_2, giving them equal opportunities to acquire their respective semaphores (Sender\_1 and Sender\_2) and attempt to send messages to the queue. The tasks would generate timestamps, send messages, and update the counters accordingly.

This priority-based scheduling ensures that Sender\_Task\_3, with a higher priority, gets executed with priority whenever it is ready to run, while Sender\_Task\_1 and Sender\_Task\_2, with the same lower priority, share the CPU fairly using round-robin scheduling. These sender tasks work concurrently and are triggered by timer callbacks. Each task waits for a specific semaphore to be released before sending a message to the queue. The semaphores act as signals to indicate when a sender task can proceed. The tasks generate timestamps for the messages they send .

**Sender Timer Callback**

The sender callback functions are called when the timer expires. The functions first acquire the corresponding semaphore, which ensures that only one task can access the queue at a time. Once the semaphore is acquired, the functions generate a random number between the lower and upper bounds, and then change the timer period to the random number. The functions also increment the corresponding sum of the timer. The sender callback functions are important because they ensure that the sender tasks are not running all the time. By randomly changing the timer period, the sender tasks are forced to wait for a period before they can send another message. This helps to prevent the sender tasks from overwhelming the receiver task.

static void SenderTimerCallback1( TimerHandle\_t xTimer )

{

xSemaphoreGive(Sender\_1); //release semaphore to act like a signal for Sender Task 1

Time\_Sender=uniform\_distribution(lower\_bound[i],upper\_bound[i]); //Change Time of Sender Callback function

sum\_sendr\_time[0]=sum\_sendr\_time[0]+Time\_Sender;

xTimerChangePeriod(xTimer1,Time\_Sender,0); //Function that changes Callback function time

}

**Receiver Task**

The receiver task is responsible for receiving messages from the queue. It does this by waiting for a signal from the timer callback function, which indicates that a message has been sent. Once the receiver task has received the signal, it will attempt to receive a message from the queue. If the message is received successfully, the receiver task will increment the number of received messages counter. If the message is not received successfully, the receiver task will print a message indicating that the message was not received.

void Receiver\_Task(void \*p) //Received Task

{

char Data\_Received[20]; //variable to store data that will be received to Queue

while(1){

if(xSemaphoreTake(Receiver, portMAX\_DELAY)){ //Semaphore waiting for signal from Timer Callback function

if(xQueueReceive(Queue,Data\_Received,0)){ //Check if receiving messages was successful

Number\_of\_Received\_Messages++; //Increment successful received messages

printf("Received %s \n",Data\_Received);

} } } }

The receiver task also has a semaphore. This semaphore is used to ensure that only one task can access the queue at a time. This is important to prevent race conditions, which can occur when multiple tasks are trying to access the same data at the same time.

By giving the receiver task a higher priority, we can ensure that it is able to receive messages from the queue as soon as possible, and that the system can continue to operate correctly. Because if the receiver task has a lower priority than the sender tasks, then the sender tasks may be able to send messages to the queue faster than the receiver task can receive them. This can lead to a backlog of messages in the queue, which can eventually cause the system to deadlock.

**Receiver Timer Callback**

function is called when the timer expires. The function first releases the Receiver semaphore, which allows the receiver task to continue running. Then, the function checks if the number of received messages is equal to 1000. If it is, the function calls the Reset\_Function() function.

static void ReciverTimerCallback( TimerHandle\_t xTimer )

{

xSemaphoreGive(Receiver); //release semaphore to act like a signal for Receiver Task 1

if(Number\_of\_Received\_Messages == 1000){

Reset\_Function(); //Call reset function if messages reached 1000 message

}

}

References :