Institute Supérieur de l'Aéronautique et de l'Espace

Master in Aerospace Engineering

Real Time Control

Real Time Control
Mutex and Semaphores

Professor:
Janette Cardoso

Group:
Jorge GALVAN LOBO
Akash SHARMA

Contents

List of Figures

List of Tables

1 Introduction

Real-time control systems are closed-loop control systems where one has a tight time window to gather data, process that data, and update the system. If the time window is missed, then the stability of the system is degraded. Real-time control of any system becomes important when the response is required within a specified time-frame. To incorporate this notion of a time-frame, a deadline is introduced for each task with each task executed sequentially by the compiler. In this project, we understand how a real time control architecture is formed and analyze the role of priority as well as different control mechanisms such as mutex and semaphore. The architecture is simplified for just 2 tasks, namely, producer and consumer. The same is shown in Figure 1.

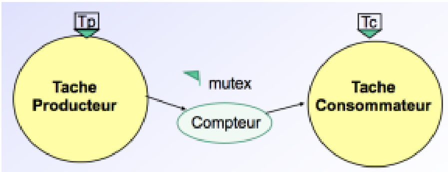

Figure 1: The control architecture provided in C code

2 Exercise 1

2.1 Question 1.1 & 1.2

The file containing the C code, namely, PeriodicTasks.c is opened in NetBeans. This file contains parameters defining the priorities and periods of the 2 tasks along with the Xenomai identifiers, exitApplicationSem and counterMutex that are responsible for task synchronisation and variable protection. The initialization of these parameters in the C code is detailed below.

Tasks
RT_TASK producerTask;
RT_TASK consumerTask;

Semaphore identifier
RT_MUTEX counterMutex;

Task Priorities and Periods
#define PRODUCER_PRIORITY 6
#define CONSUMER_PRIORITY 5
#define PRODUCER_PERIOD 10000 // Period of task tProducer, in µs
#define CONSUMER_PERIOD 200000 // Period of task tConsumer, in µs

The main features defining the Rate Monotonic Method are -

- Task scheduling is based on a fixed priority for static and critical applications that is assigned at design-time (off-line).
- Highest priority should be given to the task with the lowest period.

The above mentioned values are found to be coherent here based on the Rate Monotonic method. The function of the counterMutex is to protect the variable updation of the variable counter. Once the counterMutex is acquired by any task through the rt_mutex_acquire() function, that task is able to be executed regardless of another higher priority task being present. Other higher priority tasks can only be executed once the counterMutex is released through the rt_mutex_release() function.

2.2 Question 1.3

RTOS services -

- rt_task_inquire() & rt_task_self - This function is used to retrieve information about a given task which is given as a parameter.
  rt_task_inquire(rt_task_self(), TaskInfo); // rt_task_self : retrieves the current task

- rt_task_set_period() - Each task has to be made periodic by setting the period through this function. The period is given as a parameter while calling this function.
  rt_task_set_periodic(NULL, TM_NOW, PRODUCER_PERIOD * 1000);
- rt_task_wait_period() - Once a given task is fully executed, this function makes it wait until the period arrives before releasing it.
  rt_task_wait_period(NULL);
- rt_mutex_create() - This creates a mutex used by the tasks to synchronise.
- rt_mutex_acquire() - This function captures the mutex and holds it for a given task until it is released. It is used to protect any instructions that need to be executed within a task without being preempted.
  rt_mutex_acquire(counterMutex, TM_INFINITE); // timeout should be defined to ensure Task timing
- rt_mutex_release() - This function releases the mutex
  rt_mutex_release(counterMutex);
- rt_sem_create() - This function creates a counting semaphore
  rt_sem_create(exitApplicationSem, "exit", 0, S_FIFO); // Here, the semaphore count is set to 0 which means there are no resources to share between the tasks
- rt_task_spawn() - This function is used to create our 2 real time tasks along with their priority.
  rt_task_spawn(&producerTask, "Producer", 0 , PRODUCER_PRIORITY, T_FPU, &tProducer, NULL);
  rt_task_spawn(&consumerTask, "Consumer", 0 , CONSUMER_PRIORITY, T_FPU, &tConsumer, NULL);
- rt_task_delete() - This deletes a real time task at the end of the program run.
  rt_task_delete(&producerTask);
  rt_task_delete(&consumerTask);

forceCPUuse() - This function is called to set the duration of a particular task when it is to be executed for the given duration. The structure of the information of the task is updated until the execution time reaches the total duration of the task.

tProducer() - This function, firstly, makes the PRODUCER task periodic by using the previously defined period PRODUCER_PERIOD. Then, the shared variable is incremented in value after the task acquires the mutex which provides protection during the variable updation. It finally releases the mutex and calls the forceCPUuse() function.

tConsumer() - The tConsumer() function, similar to the tProducer() function makes the CONSUMER task periodic by using the period CONSUMER_PERIOD. Then, it acquires the mutex and prints the shared variable which is followed by releasing the mutex and calling the forceCPUuse() function. Lastly, it also checks whether the timer has gone past the execution time. If this is true, then the exit semaphore signal is given to the pending main task.

2.3 Question 1.4

Code Output -

```
Creating the application mutexes ....
 Creating the application tasks ....
 Tasks .. created and launched

counter= 20 time :201203.000
counter= 40 time :401137.000
counter= 60 time :601094.000
counter= 80 time :801670.000
counter= 100 time :1001317.000
counter= 120 time :1202113.000
counter= 140 time :1401002.000
counter= 160 time :1601023.000
counter= 180 time :1801025.000
counter= 200 time :2001172.000
counter= 220 time :2201002.000
counter= 240 time :2401025.000
counter= 260 time :2601374.000
counter= 280 time :2801373.000
counter= 300 time :3001029.000
counter= 320 time :3201003.000
counter= 340 time :3401008.000
counter= 360 time :3601042.000
counter= 380 time :3801217.000
counter= 400 time :4001001.000
counter= 420 time :4201115.000
counter= 440 time :4401198.000
counter= 460 time :4601100.000
counter= 480 time :4801007.000
counter= 500 time :5001365.000
counter= 520 time :5201006.000
counter= 540 time :5401074.000
counter= 560 time :5601003.000
counter= 580 time :5801290.000
counter= 600 time :6001402.000
counter= 620 time :6201426.000
counter= 640 time :6401051.000
counter= 660 time :6601038.000
counter= 680 time :6801056.000
counter= 700 time :7001656.000
counter= 720 time :7201747.000
counter= 740 time :7401208.000
counter= 760 time :7601830.000
counter= 780 time :7801022.000
counter= 800 time :8001802.000
counter= 820 time :8201022.000
counter= 840 time :8401780.000
counter= 860 time :8601680.000
counter= 880 time :8801029.000
counter= 900 time :9001030.000
counter= 920 time :9201034.000
counter= 940 time :9401042.000
counter= 960 time :9601861.000
counter= 980 time :9801088.000
counter= 1000 time :10001061.000
 total Duration =  10056082 µs  ( 10.056082 s )
Task name : Producer, Priority : 6, Exectime MP en µs 1020176
Commutations P/S : 0 , Mode : 300184 , Context switches: 1006
Task name : Consumer, Priority : 5, Exectime MP en µs 2500500
Commutations P/S : 0 , Mode : 300184 , Context switches: 301
 Deleting the tasks ....
 Deleting the mutexes ....
 Application ..... finished --> exit

RUN FINISHED; valeur renvoyée 0 ; real time: 10s; user: 0ms; system: 0ms
```

Figure 2: first output

PRODUCER response time = 1.02s
CONSUMER response time = 2.5s

2.4 Question 1.5

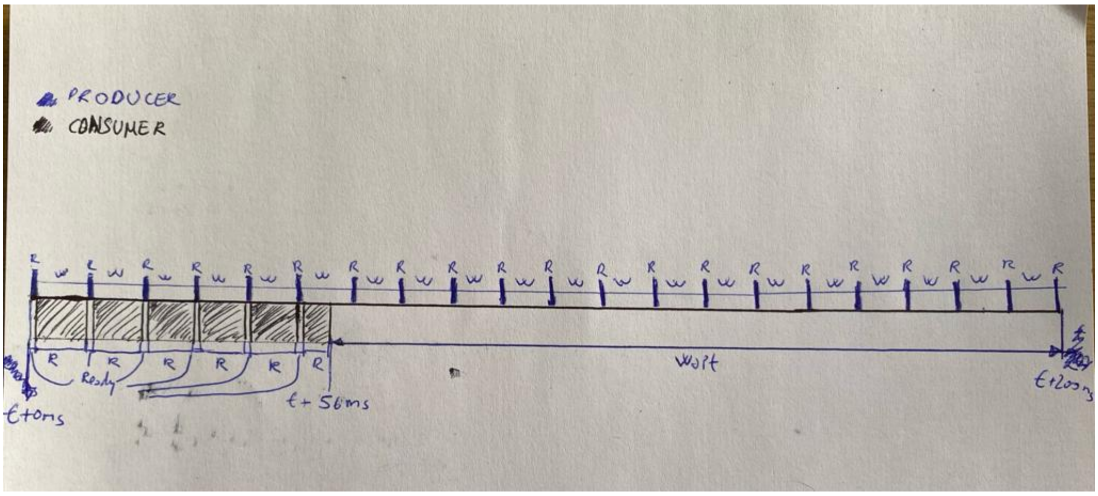

Figure 3: Time Diagram - nominal case

Time(ms)	Producer	Consumer
0	Active	Ready
1	Wait	Active
10	Active	Ready
11	Wait	Active
20	Active	Ready
21	Wait	Active
30	Active	Ready
31	Wait	Active
40	Active	Ready
41	Wait	Active
50	Active	Ready
51	Wait	Active
56	Wait	Wait

Table 1: Tasks States

2.5 Question 1.6

In the time duration from t=56ms until t=200ms, the PRODUCER task is executed every 10ms as expected while the CONSUMER waits for its period to be over. The file prodConsText is then run on Cheddar and the simulation time diagram is compared with the hand-drawn one.

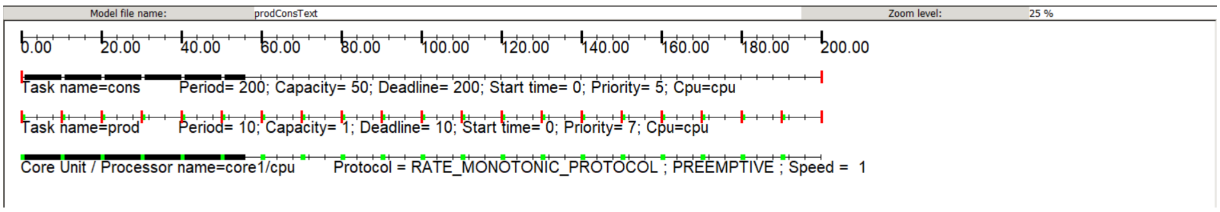

Figure 4: Cheddar Time Diagram

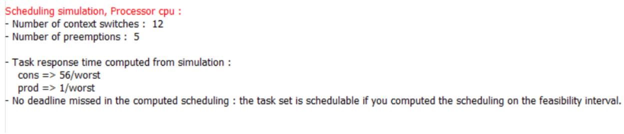

Figure 5: Cheddar values

2.6 Question 1.7

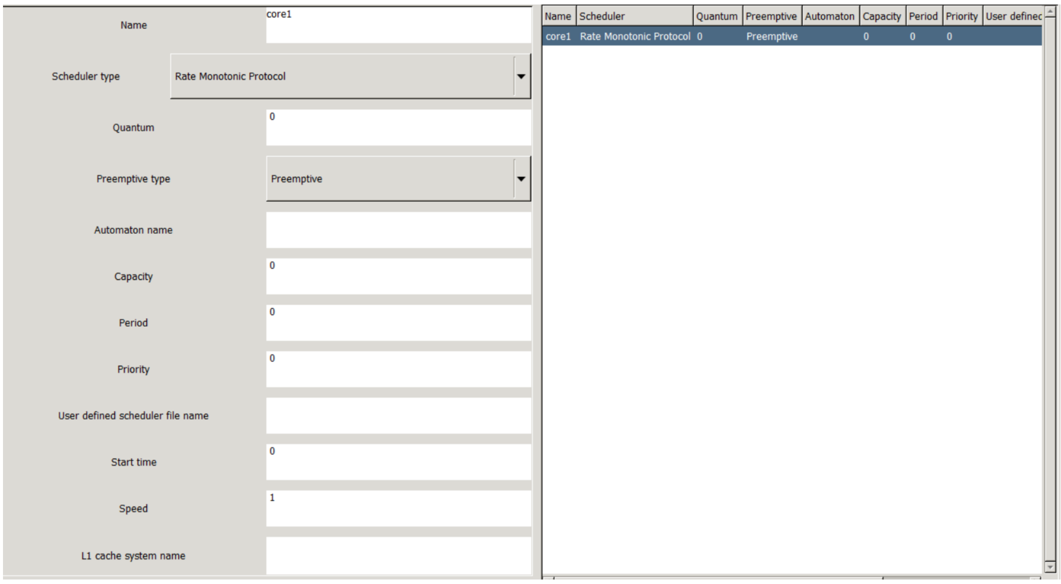

Figure 6: Scheduler used

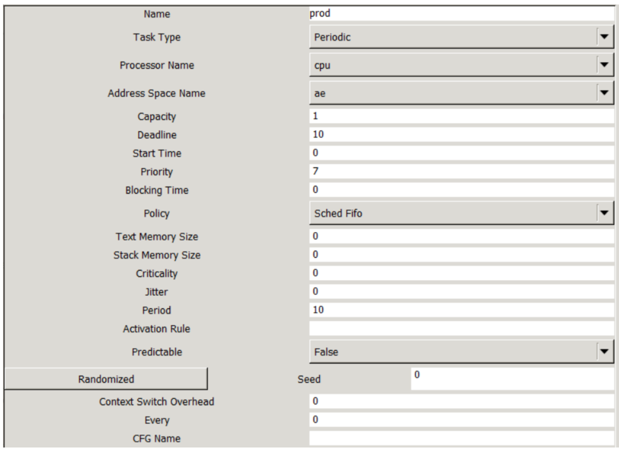

Figure 7: Producer defined in Cheddar

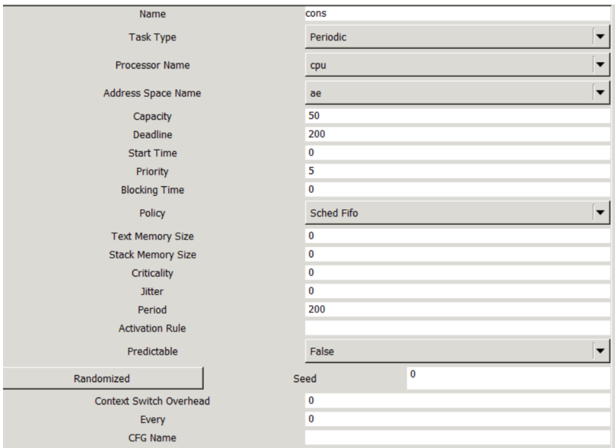

Figure 8: Consumer defined in Cheddar

3 Exercise 2

The execution of the c code generates a predictable result. Therefore, by analysing it, the output of the program can be understood. The part of the code that explains the operations to be done in the tasks will allow this justification.(9)(10)

```c
void tProducer()
// Periodic task
// Output : data counter is incremented
// Tasks functions (entry points)
{
  rt_task_set_periodic(NULL, TM_NOW, PRODUCER_PERIOD * 1000);

  while( 1 ) {
    rt_task_wait_period(NULL);
    rt_mutex_acquire(&counterMutex, TM_INFINITE); // timeout should be defined to ensure Task timing
    counter++; // The updated in this variable is protected by counterMutex
    rt_mutex_release(&counterMutex);

    forceCPUuse(PRODUCER_CPUuse); // Force CPU use while producing data

  }
}
```

Figure 9: tproducer()

```
void tConsumer()
// Periodic task
// Input : data counter
// Display time  and value of counter
// Tasks functions (entry points)
{
  int c;
  // calculate the end of execution
  RTIME endtime = rt_timer_read() +  (APP_DURATION * ONE_SECOND); // Time in ns

  rt_task_set_periodic(NULL, TM_NOW, CONSUMER_PERIOD * 1000);

  while( 1 ) {
    rt_task_wait_period(NULL);
    rt_mutex_acquire(&counterMutex, TM_INFINITE); // timeout should be defined to ensure Task timing
    // The reading of 'counter' is protected by counterMutex and can be done safely
    c=counter;
    rt_mutex_release(&counterMutex);
    rt_printf("counter= %d time :%8.3f  \n\r",c,(double)((rt_timer_read()-t0)/1000));

    forceCPUuse(CONSUMER_CPUuse); // Force CPU use while consuming data
    if( rt_timer_read() >= endtime ) // check if execution must finish
      rt_sem_v(&exitApplicationSem); // give exit semaphore to main task

  }
}
```

Figure 10: tconsumer()

The output of the execution of the code returns counter=20 and time=20xxxx. In particular this output is produced by (11)

```
rt_printf("counter= %d time :%8.3f  \n\r",c,(double)((rt_timer_read()-t0)/1000));
```

Figure 11: rt_print()

By observing the time diagram from exercise 1, the output from the program can be reconstructed. The counter corresponds to how many executions of producer() the code has performed before a particular consumer execution. Therefore, as there are 20 producers in every consumer cycle, it makes sense that the output advances in blocks of 20. Furthermore,the program initializes both functions at t=0. However, by t=10, producer is already augmenting the counter. Consumer, on the other hand, must wait to t=200 to ask for the first output, as both functions must run for 1 period before they can operate nominally. In conclusion, the previous execution of producer explains the non zero counter in the first output, and the period of the tasks explains the counter increment. In regards of the time, the extra execution time present in the output can be explained by observing the code, as there are indeed some lines of code besides the actual output and the counter management. This lines will require a small, yet noticeable, CPU usage. This is usage, is the one observed in the output, and its order of magnitude is reasonable for the length of the code and their significance.

4 Exercise 3

4.1 Question 3.1

The code is executed with CONSUMER_CPUuse=50000 and CONSUMER_CPUuse=150000. The logs of the console are stored and displayed in (12)(13)

```
Creating the application mutexes ....
 Creating the application tasks ....
 Tasks .. created and launched

counter= 20 time :201203.000
counter= 40 time :401137.000
counter= 60 time :601094.000
counter= 80 time :801670.000
counter= 100 time :1001317.000
counter= 120 time :1202113.000
counter= 140 time :1401002.000
counter= 160 time :1601023.000
counter= 180 time :1801025.000
counter= 200 time :2001172.000
counter= 220 time :2201002.000
counter= 240 time :2401025.000
counter= 260 time :2601374.000
counter= 280 time :2801373.000
counter= 300 time :3001029.000
counter= 320 time :3201003.000
counter= 340 time :3401008.000
counter= 360 time :3601042.000
counter= 380 time :3801217.000
counter= 400 time :4001001.000
counter= 420 time :4201115.000
counter= 440 time :4401198.000
counter= 460 time :4601100.000
counter= 480 time :4801007.000
counter= 500 time :5001365.000
counter= 520 time :5201006.000
counter= 540 time :5401074.000
counter= 560 time :5601003.000
counter= 580 time :5801290.000
counter= 600 time :6001402.000
counter= 620 time :6201426.000
counter= 640 time :6401051.000
counter= 660 time :6601038.000
counter= 680 time :6801056.000
counter= 700 time :7001656.000
counter= 720 time :7201747.000
counter= 740 time :7401208.000
counter= 760 time :7601830.000
counter= 780 time :7801022.000
counter= 800 time :8001802.000
counter= 820 time :8201022.000
counter= 840 time :8401780.000
counter= 860 time :8601680.000
counter= 880 time :8801029.000
counter= 900 time :9001030.000
counter= 920 time :9201034.000
counter= 940 time :9401042.000
counter= 960 time :9601861.000
counter= 980 time :9801088.000
counter= 1000 time :10001061.000
 total Duration =  10056082 µs  ( 10.056082 s )
Task name : Producer, Priority : 6, Exectime MP en µs
1020176
Commutations P/S : 0 , Mode : 300184 , Context switches:
1006
Task name : Consumer, Priority : 5, Exectime MP en µs
2500500
Commutations P/S : 0 , Mode : 300184 , Context switches:
301
 Deleting the tasks ....
 Deleting the mutexes ....
 Application ..... finished --> exit


RUN FINISHED; valeur renvoyée 0 ; real time: 10s; user:
0ms; system: 0ms
```

Figure 12: nominal console output

```
Creating the application mutexes ....
 Creating the application tasks ....
 Tasks .. created and launched

counter= 20 time :201132.000
counter= 40 time :401811.000
counter= 60 time :601036.000
counter= 80 time :801067.000
counter= 100 time :1001088.000
counter= 120 time :1201036.000
counter= 140 time :1401031.000
counter= 160 time :1601110.000
counter= 180 time :1801018.000
counter= 200 time :2001072.000
counter= 220 time :2201049.000
counter= 240 time :2401643.000
counter= 260 time :2601783.000
counter= 280 time :2801013.000
counter= 300 time :3001081.000
counter= 320 time :3201000.000
counter= 340 time :3401048.000
counter= 360 time :3601045.000
counter= 380 time :3801041.000
counter= 400 time :4001035.000
counter= 420 time :4201571.000
counter= 440 time :4401040.000
counter= 460 time :4601073.000
counter= 480 time :4801670.000
counter= 500 time :5001008.000
counter= 520 time :5201024.000
counter= 540 time :5401018.000
counter= 560 time :5601024.000
counter= 580 time :5801103.000
counter= 600 time :6001044.000
counter= 620 time :6201072.000
counter= 640 time :6401037.000
counter= 660 time :6601047.000
counter= 680 time :6801752.000
counter= 700 time :7001075.000
counter= 720 time :7201658.000
counter= 740 time :7401035.000
counter= 760 time :7601081.000
counter= 780 time :7801042.000
counter= 800 time :8001001.000
counter= 820 time :8201067.000
counter= 840 time :8401045.000
counter= 860 time :8602258.000
counter= 880 time :8802191.000
counter= 900 time :9001002.000
counter= 920 time :9201037.000
counter= 940 time :9401070.000
counter= 960 time :9601049.000
counter= 980 time :9801045.000
counter= 1000 time :10002167.000
 total Duration =  10168201 µs  ( 10.168201 s )
Task name : Producer, Priority : 6, Exectime MP en µs
1026254
Commutations P/S : 0 , Mode : 300184 , Context switches:
1017
Task name : Consumer, Priority : 5, Exectime MP en µs
7500440
Commutations P/S : 0 , Mode : 300184 , Context switches:
851
 Deleting the tasks ....
 Deleting the mutexes ....
 Application ..... finished --> exit


RUN FINISHED; valeur renvoyée 0 ; real time: 10s; user:
0ms; system: 0ms
```

Figure 13: longer consumer execution console output

The differences between the outputs are quite apparent when observing the last lines of each log. In particular, the attention should be placed upon the context switches. In the nominal case, around 1000 switches are assigned to Producer and 300 switches are assigned to Consumer. In the case with CONSUMER_CPUuse=150000, the switches of Producer stay more or less constant, but the switches of Consumer get almost tripled. This result is reasonable. Given the lengthened execution time of Consumer, it will be interrupted by Producer more times. AS the priorities have not changed, Consumer will have to be slotted in between producers. As Consumer now takes three times more to finish, it will be switched, and preempted, roughly three times more.

4.2 Question 3.2

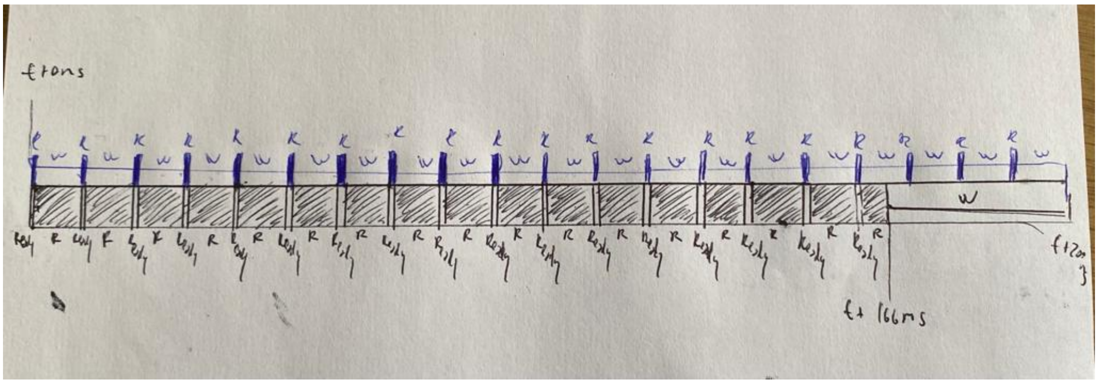

Figure 14: longer consumer execution diagram

The presented diagram corresponds to the task scheduling with a more time exigent Consumer Task. The result is coherent with both the expectation and the Netbeam console logs. This is, that the Consumer task gets dragged out for longer as it is interrupted by producer during 150ms instead of 50ms.

5 Exercise 4

5.1 Question 4.1

The diagrams corresponding to the inverted priorities are created by hand. The main principle followed is the monotonic scheduling. However, missing deadlines will be allowed so as to visualize the effects of wrongly assigned priorities.

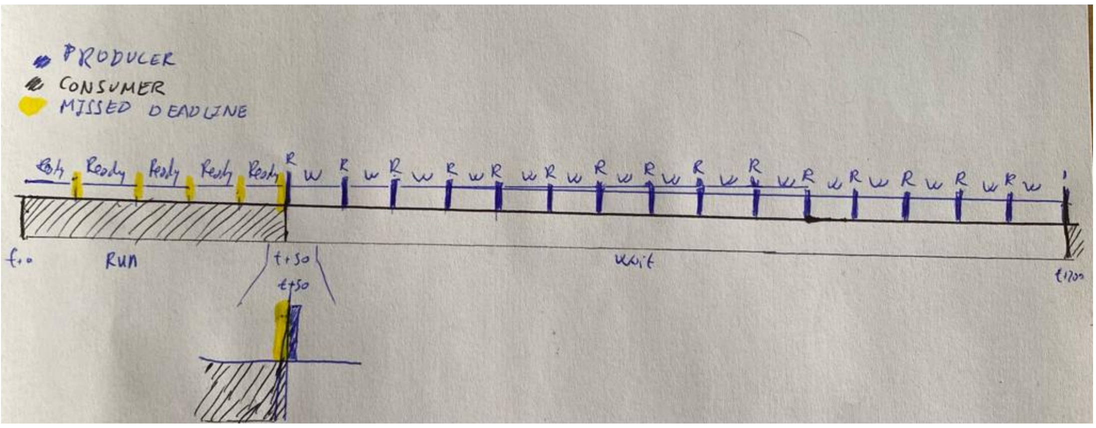

Figure 15: priority inverted diagram

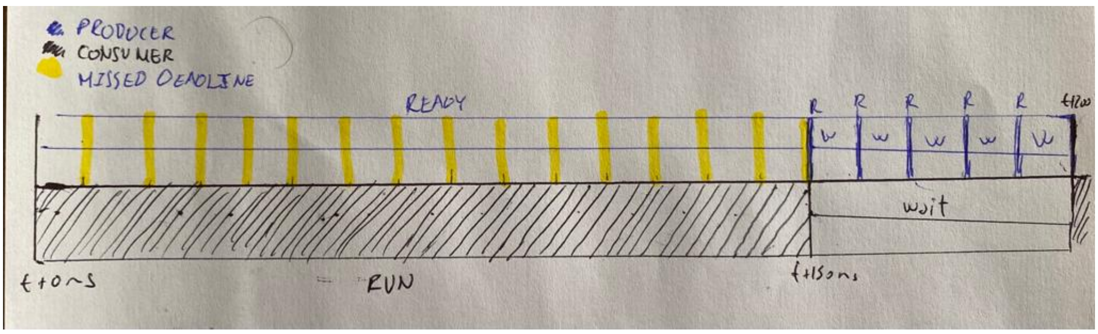

Figure 16: longer consumer execution priority inverted diagram

5.2 Question 4.2

Having observed the diagrams, the execution logs should confirm the results. They are obtained and displayed in (17)(18)

```
Consumer_capacity = 50,000
Creating the application mutexes ....
 Creating the application tasks ....
 Tasks .. created and launched

counter= 19 time :200058.000
counter= 34 time :400073.000
counter= 49 time :600057.000
counter= 64 time :800402.000
counter= 79 time :1000777.000
counter= 94 time :1200054.000
counter= 109 time :1400374.000
counter= 124 time :1600108.000
counter= 139 time :1800076.000
counter= 154 time :2000750.000
counter= 169 time :2200086.000
counter= 184 time :2400056.000
counter= 199 time :2600020.000
counter= 214 time :2800059.000
counter= 229 time :3000006.000
counter= 244 time :3200491.000
counter= 259 time :3400768.000
counter= 274 time :3600716.000
counter= 289 time :3800020.000
counter= 304 time :4000024.000
counter= 319 time :4200015.000
counter= 334 time :4400720.000
counter= 349 time :4600098.000
counter= 364 time :4800633.000
counter= 379 time :5000019.000
counter= 394 time :5200016.000
counter= 409 time :5400028.000
counter= 424 time :5600076.000
counter= 439 time :5800091.000
counter= 454 time :6000073.000
counter= 469 time :6200065.000
counter= 484 time :6400679.000
counter= 499 time :6600756.000
counter= 514 time :6800392.000
counter= 529 time :7000762.000
counter= 544 time :7200192.000
counter= 559 time :7400019.000
counter= 574 time :7600969.000
counter= 589 time :7800050.000
counter= 604 time :8000021.000
counter= 619 time :8200005.000
counter= 634 time :8400492.000
counter= 649 time :8600440.000
counter= 664 time :8800026.000
counter= 679 time :9000039.000
counter= 694 time :9200199.000
counter= 709 time :9400441.000
counter= 724 time :9600020.000
counter= 739 time :9800045.000
counter= 754 time :10000016.000
 total Duration =  10051034 µs  ( 10.051034 s )
Task name : Producer, Priority : 6, Exectime MP en µs
793682
Commutations P/S : 0 , Mode : 300184 , Context switches:
756
Task name : Consumer, Priority : 7, Exectime MP en µs
2503347
Commutations P/S : 0 , Mode : 300184 , Context switches:
51
 Deleting the tasks ....
 Deleting the mutexes ....
 Application ..... finished --> exit


RUN FINISHED; valeur renvoyée 0 ; real time: 10s; user:
0ms; system: 0ms
```

Figure 17: priority inverted console output

```
Consumer_capacity = 150,000
 Creating the application mutexes ....
 Creating the application tasks ....
 Tasks .. created and launched

counter= 19 time :200283.000
counter= 24 time :401517.000
counter= 29 time :600587.000
counter= 34 time :800043.000
counter= 40 time :1000878.000
counter= 45 time :1201554.000
counter= 50 time :1400037.000
counter= 55 time :1600750.000
counter= 60 time :1801210.000
counter= 65 time :2000040.000
counter= 70 time :2200048.000
counter= 75 time :2400498.000
counter= 80 time :2600049.000
counter= 85 time :2800065.000
counter= 90 time :3001161.000
counter= 95 time :3200040.000
counter= 100 time :3400040.000
counter= 106 time :3600871.000
counter= 111 time :3800050.000
counter= 116 time :4000005.000
counter= 121 time :4200038.000
counter= 126 time :4400039.000
counter= 131 time :4600930.000
counter= 136 time :4800057.000
counter= 141 time :5000040.000
counter= 146 time :5201437.000
counter= 151 time :5400038.000
counter= 156 time :5600092.000
counter= 161 time :5800074.000
counter= 166 time :6000054.000
counter= 171 time :6200821.000
counter= 176 time :6400041.000
counter= 181 time :6600049.000
counter= 186 time :6801072.000
counter= 191 time :7000092.000
counter= 196 time :7201013.000
counter= 202 time :7400433.000
counter= 207 time :7600038.000
counter= 212 time :7800075.000
counter= 217 time :8000092.000
counter= 222 time :8200040.000
counter= 227 time :8401050.000
counter= 232 time :8600036.000
counter= 237 time :8800044.000
counter= 242 time :9000062.000
counter= 247 time :9200039.000
counter= 252 time :9400935.000
counter= 257 time :9600049.000
counter= 262 time :9800100.000
counter= 267 time :10000035.000
 total Duration =  10151071 µs  ( 10.151072 s )
Task name : Producer, Priority : 6, Exectime MP en µs
271522
Commutations P/S : 0 , Mode : 300184 , Context switches:
269
Task name : Consumer, Priority : 7, Exectime MP en µs
7501812
Commutations P/S : 0 , Mode : 300184 , Context switches:
51
 Deleting the tasks ....
 Deleting the mutexes ....
 Application ..... finished --> exit


RUN FINISHED; valeur renvoyée 0 ; real time: 10s; user:
0ms; system: 0ms
```

Figure 18: longer consumer execution priority inverted console output

By observing the output, we can extract some conclusions. Firstly, the jumps of either execution can be explained given the execution priority. The counter is only incremented by producer. Therefore, the jumps between counter displays correspond to the amount of producer tasks successfully executed between consumer tasks. In the nominal case, the producer has the higher priority, it therefore is executed successfully the 20 times it is supposed to between each consumer execution. However, by inverting the priorities, Consumer will block out the execution of producer. The longer it takes to fulfill Consumer, the more Producers that will be ignored. The more producer tasks ignored,the less counter increments executed. In the first case(17), Consumer blocks out 5 producer tasks, therefore the counter is incremented by the maximum, 20, minus those 5. Ergo, 15 each time. In the second case(18), Consumer blocks out 15 producer tasks. Thus, the 5 counter increment. Furthermore, the fact that the context switches remain constant for Consumer, indicate that it is indeed being executed with a higher priority. The context switches of Producer indicate that less producer tasks are being executed the higher the duration of consumer. The higher the duration, the less switches, as more tasks are simply block form execution.

6 Exercise 5

rt sem create (RT SEM sem, const char name, unsigned long icount, int mode)

This function creates a counting semaphore. A semaphore is an integer variable, that is initialized with the number of resources present in the system and is used for process synchronization. It uses two functions to change the value of S i.e. wait() and signal(). Both these functions are used to modify the value of semaphore but the functions allow only one process to change the value at a particular time i.e. no two processes can change the value of semaphore simultaneously.
In counting semaphores, the semaphore variable is initialized with the number of resources available. After that, whenever a process needs some resource, then the wait() function is called and the value of the semaphore variable is decreased by one. The process then uses the resource and after using the resource, the signal() function is called and the value of the semaphore variable is increased by one. So, in case the value of the semaphore variable goes to 0 i.e all the resources are taken by the process and there is no resource left to be used and if some other process wants to use resources, then that process has to wait for its turn. In this way, we achieve the process synchronization.
Here, the initial value of icount = 0. This means that essentially there are no resources to share. The main() function waits for this semaphore signal which is only released when the execution time is finished i.e. the timer has reached the APP_DURATION. Once it receives the signal, it allows it to delete all the tasks and mutexes before closing the application.

6.1 Question 5.1

A critical section of a task is a section of code that accesses a shared resource. This section is protected by the mutex when the resource is utilized. The file PeriodicTasksCS.c is run which takes into account the critical sections by forcing execution before, during and after this section is run. The bodies of tProducer() and tConsumer are shown in (19) & (20) respectively.

```
void tProducer()
// Periodic task
// Output : data counter is incremented
// Tasks functions (entry points)
{
  rt_task_set_periodic(NULL, TM_NOW, PRODUCER_PERIOD * 1000);

  while( 1 ) {
    rt_task_wait_period(NULL);
    forceCPUuse(PRODUCER_StartCS);  // Execution before critical section
    // If mutex is already taken, the task is blocked (waits for this resource)
    rt_mutex_acquire(&counterMutex, TM_INFINITE);
    counter++; // The updated in this variable is protected by counterMutex
    forceCPUuse(PRODUCER_durationCS); // Execution duration of critical section
    rt_mutex_release(&counterMutex);
    forceCPUuse(PRODUCER_CPUuse-PRODUCER_durationCS-PRODUCER_StartCS); // Execution after critical section

  }
}
```

Figure 19: PRODUCER body of code

```c
void tConsumer()
// Periodic task
// Input : data counter
// Display time  and value of counter
// Tasks functions (entry points)
{
  int c;
  // calculate the end of execution
  RTIME endtime = rt_timer_read() +  (APP_DURATION * ONE_SECOND); // Time in ns

  rt_task_set_periodic(NULL, TM_NOW, CONSUMER_PERIOD * 1000);

  while( 1 ) {
    rt_task_wait_period(NULL);
    // A task can be preempted by another task with higher priority
    forceCPUuse(CONSUMER_StartCS); // Execution before critical section
    rt_mutex_acquire(&counterMutex, TM_INFINITE); // timeout should be defined to ensure Task timing
    forceCPUuse(CONSUMER_durationCS); // Execution duration of critical section
    // The reading of 'counter' is protected by counterMutex and can be done safely
    c=counter;
    // Once this task T releases the mutex, if there is a task T' (with higher
    // priority) blocked by this mutex, T' will be Ready and preempt T
    rt_mutex_release(&counterMutex);
    // The printing occurs after the release: if preempted by T', this task will
    // print only after completion of T'!
    rt_printf("counter= %d time :%8.3f  \n\r",c,(double)((rt_timer_read()-t0)/1000));
    forceCPUuse(CONSUMER_CPUuse-CONSUMER_StartCS-CONSUMER_durationCS); // Execution after critical section
    if( rt_timer_read() >= endtime ) // check if execution must finish
      rt_sem_v(&exitApplicationSem); // give exit semaphore to main task

  }
}
```

Figure 20: CONSUMER body of code

The new variables *_StartCS and *_durationCS are defined as shown below.

#define PRODUCER_StartCS 1000 // µs
#define PRODUCER_durationCS 7000 // µs
#define CONSUMER_StartCS 50000 // µs
#define CONSUMER_durationCS 20000 // µs

6.2 Question 5.2

The models prodCons and prodConsCriticalSectionSmall are run and the results are as shown below. The prodCons result is similar to the nominal case but with all values multiplied with 10. The prodconsCriticalSectionSmall result shows that since the critical section is small (only 20ms and 7ms), it does not interfere with the execution of the tasks.

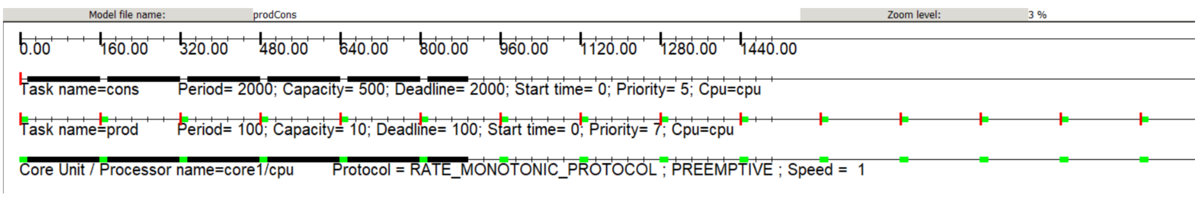

Figure 21: prodCons

```
Scheduling simulation, Processor cpu :
- Number of context switches :  12
- Number of preemptions :  5

- Task response time computed from simulation :
    cons => 560/worst
    prod => 10/worst
- No deadline missed in the computed scheduling : the task set is schedulable if you computed the scheduling on the feasibility interval.
```

Figure 22: prodCons Data

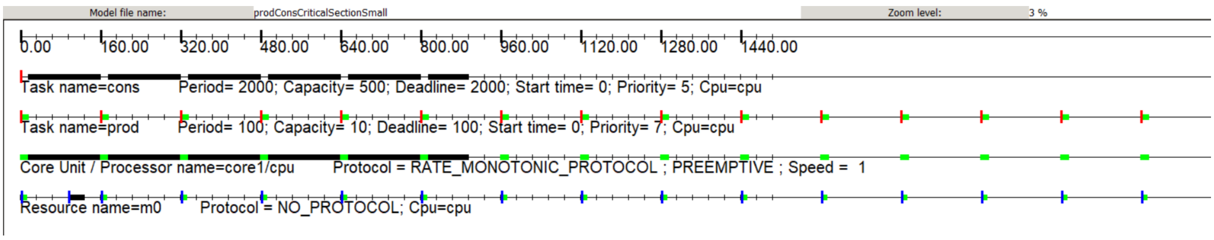

Figure 23: prodConsCriticalSectionSmall

```
Scheduling simulation, Processor cpu :
- Number of context switches :  12
- Number of preemptions :  5

- Task response time computed from simulation :
    cons => 560/worst
    prod => 10/worst
- No deadline missed in the computed scheduling : the task set is schedulable if you computed the scheduling on the feasibility interval.
```

Figure 24: prodConsCriticalSectionSmall Data

6.3 Question 5.3

After this, the critical section of the consumer is changed to a larger value of duration = 400ms and the C code is run.

```
 Creating the application mutexes ....
 Creating the application tasks ....
 Tasks .. created and launched

counter= 20 time :2480090.000
counter= 38 time :4480509.000
counter= 56 time :6480033.000
 total Duration =  6540046 µs  ( 6.540046 s )
Task name : Producer, Priority : 7, Exectime MP en µs 591482
Commutations P/S : 0 , Mode : 300184 , Context switches: 60
Task name : Consumer, Priority : 5, Exectime MP en µs 1500044
Commutations P/S : 0 , Mode : 300184 , Context switches: 13
 Deleting the tasks ....
 Deleting the mutexes ....
 Application ..... finished --> exit

RUN FINISHED; valeur renvoyée 0 ; real time: 6s; user: 0ms; system: 0ms
```

Figure 25: C code result for 400ms consumer critical section duration

The same result is simulated in Cheddar. As we can see, some deadlines have not been met. This is due to the fact that the critical section duration surpasses some periods of the PRODUCER task. This disrupts the entire sequence of the process of the task execution.

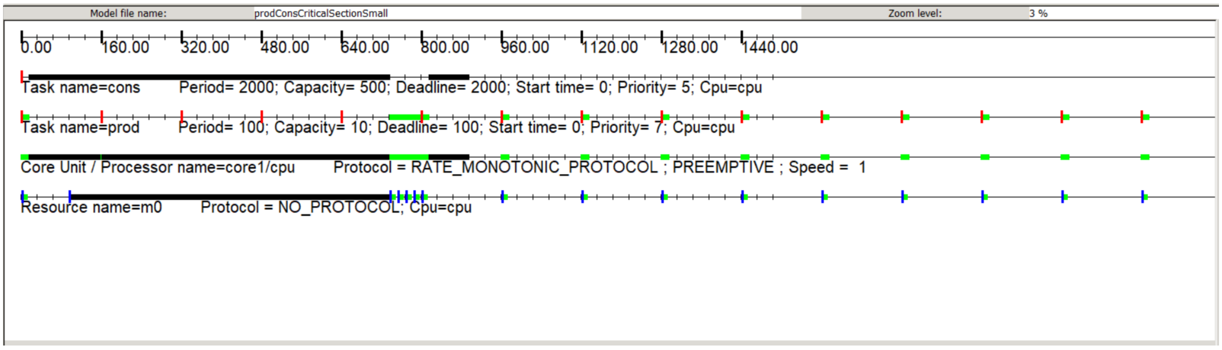

Figure 26: prodConsCriticalSectionBig Cheddar time diagram

```
Scheduling simulation, Processor cpu :
- Number of context switches :  6
- Number of preemptions :  3

- Task response time computed from simulation :
   cons => 560/worst
   prod => 370/worst ,  missed its deadline (absolute deadline =  200 ; completion time =  470),  missed its deadline (absolute deadline =  300 ; completion time =  480),  missed its deadline (absolute deadline =  400 ; completion time =  490)
- Some task deadlines will be missed : the task set is not schedulable.
```

Figure 27: prodConsCriticalSectionBig Cheddar time diagram logs