**CG2271 Real Time Operating Systems**

**Term Assignment II**

**Report**

|  |  |
| --- | --- |
| Member 1 Matric: A0077863N | Member 1 Name: Song Yangyu |
| Member 2 Matric: A0078040M | Member 2 Name: Li Ruize |

*Please keep to under 4 pages. Anything past page 4 will not be marked. So make judicious use of this space. For example you won’t want to be dumping large chunks of code. ☺ You may delete all italicized instructions.*

1. The OSSignal Structure

*Talk about why you think RTOS often have just one common structure for all mechanisms, the advantages, disadvantages. Talk about the design of your OSSignal structure and how you’ve designed it to satisfy the needs of semaphores and queues while minimizing its size.*

*RTOS often have one common structure for all mechanism:*

*Advantage:*

1. *The code for creating one mechanism can be reused by other mechanisms. For example, semaphore requires a queue; therefore it can take the implementation of queue directly.*
2. *One is possible to use array to store different mechanism. This can allow rather fast access to the element*

*Disadvantage:*

1. *Wasting storage – there’re some components in the common structure may not be used by some mechanisms. For example, the queue size might be large, while the queue of semaphore would be small; in that case, one would possibly take the size of the larger one (another way is to use pointer; but it’s quite unsafe)*
2. *Ambiguous naming – like in the semaphore, one would need a counter; while in queue, one need the total number of elements. In that case, one would have to find some name in the middle of the two, like “num” , but this is still quite ambiguous.*
3. *Difficulty in adjusting to the correct structure (like different types of array) when coding. For example, the size of a queue element can be variable, but for semaphore the element size is fixed; therefore using a queue structure would cause more trouble in coding*

*Structure of the OSSignal:*

typedef struct osstruct

{

short type;

int num; // the current # of elements in the queue; or the counter of the semaphore

int esize; // size of each element

int size; // the total size of the queue

void \*start; // the starting location of the queue

void \*head;// head of the queue

} OSSignal;

*The comments are quite self-explanatory. Note that when using an semaphore, the negative value of semaphore is the number of elements currently being blocked because of SemaPend; while the positive value of semaphore is the current.*

*In order to save space, the “start” is the starting address of the queue dynamically allocated; and the “head” is the address of the first element of the queue; note that it should be shifted forward after de-queue operation.*

1. Implementing the Semaphores

*Talk about your semaphore implementation. For example how do you keep track of which tasks are waiting on a semaphore? Do you put ALL tasks into the READY queue or only the highest priority? Etc.*

*Cut and paste code* ***\*judiciously\****  *to help explain your points.*

*In “OSmakeSema” function, I’ll first allocate space for semaphore and check if the space-allocation is successful, and then initialize:*

sem->type = SEMAPHORE\_TYPE;

sem->num = initval; // the (-num) is qsize; when num >= 0, qsize = 0

sem->esize = sizeof(int); // each node would store a task id

sem->size = OS\_NUM\_TASKS; // the maximum is the number of tasks system has

sem->start = calloc(sem->size,sem->esize);

*Then I’ll check if the calling of* calloc *is successful, and initialize: sem->head= sem->start;*

*In “OSSemPend” function:*

*I first call “enterCritical” to disable interrupt because the input sema is shared variable.*

*Then I check if the semaphore has been correctly initialized, and if the type is semaphore*

*Then I check if the counter <= 0 🡪*

*If no, simply decrease the counter; enable interrupt, return;*

*If yes, then put the current task id into the queue, and set the current task in the task table to be blocked (so that the findNextTask function would not take this task), then decrease the counter, then swap to another task by calling “OSSwapTask”;*

*In “OSSemPost” function, similar to OSSemPend, I’ll disable interrupt and make appropriate checking; then check if the counter < 0:*

*If yes, then simply decrease the counter, enable interrupt, return;*

*If no, then pop the element from the queue, increase the “head” (address of the first element in the queue), and then decrease the popped task’s “sem\_counter” (so that the findNextTask function would know that the block condition of this task has been canceled).*

1. Implementing Queues

*Talk about your Queue implementation.*

*Cut and paste code* ***\*judiciously\****  *to help explain your points.*

*The queue would make use of two semaphores: q\_sem\_empty and q\_sem\_full, for the blocking of task when queue is empty/full.*

*In OSmakeQ function:*

*I’ll first check if the two semaphore used here has been initialized, if not, initialize.*

*Then I’ll allocate the space for the queue structure; then assign the initial value:*

que->type = QUEUE\_TYPE;

que->num = 0; // initially empty

que->esize = esize; // store the task id

que->size = qsize; // the maximum is the number of tasks system has

que->start = calloc(que->size,que->esize);

*Same as semaphore, I’ll also check if calloc is successful, and assign: que->head = que->start.*

*In OSenq, inside a while loop, I’ll check if the structure is queue, then check if the queue is full. If yes, , I’ll call OSSemPend(q\_sema\_full) so that the task would be keep blocked until queue is no longer full;*

*Then I’ll put in the element and increase the number of elements inside the queue, and post to q\_sema\_empty just in case the queue is empty*

*Similiarly in OSdeq, in a while loop I’ll check if the queue is empty. If yes I’ll do OSSemPend(q\_sema\_empty).*

*Then I’ll save the value of the head element (which stores the address to the first element currently in the queue), increase the head element by the size of one element, and then return the saved head element.*

*After this, I’ll OSSemPost(q\_sema\_full), just in case the queue is full preciously.*

1. The Demo Program

*Talk about your program Assg2.c*, *and the creative things that you have done with it. If you have used hardware attach a short video or link to your video of your hardware running*.

The *Assg2.c* has three tasks. The first two tasks would increase the global counter, and put it into the queue every 10 counts; then the third task would pop from the queue and send it to the computer through serial port.

void task1(void \*p)

{

while(1)

{

enterCritical();

variable++;

if(variable % 10 == 0){

OSenq(queue,&variable);

} else if(variable == MAXINT){

variable = 0;

}

leaveCritical();

}

}

void task2(void \*p)

{

while(1)

{

enterCritical();

variable++;

if(variable % 10 == 0){

OSenq(queue,&variable);

} else if(variable == MAXINT){ // avoid overflow

variable = 0;

}

leaveCritical();

}

}

void task3(void \*p)

{

void \*addr;

int res,i;

while(1)

{

if(queue->size > 0){

enterCritical();

addr = OSdeq(queue);

res = \*(int\*)addr;

sprintf(sayBuf,"r:%d \n\r",res);

say(sayBuf);

\_delay\_ms(20);

leaveCritical();

}

}

}

Nothing creative done…