Test on August 2008

These are the methods, variables and definitions that you may use throughout home and class exercises :

Definitions:

#define INIFINITE_TIMEOUT 0xFFFFFFFF
#define POLICY_FIFO 0x1
#define POLICY_PRIORITY 0x2
#define BLOCK_IF_FAIL 0x1
#define ERROR_IF_FAIL 0x2
#define MAX_PRIORITY 50
#define MIN_PRIORITY 250

Methods:

Mutex:

SemMCreate (&Sem, policy), SemMTake (&Sem, timeoutvalue, shouldBlockIfFail), SemMGive (&Sem)

Inversion Safe Mutex

SemMPISafeCreate (&Sem), SemMPISafeTake (&Sem, timeoutvalue, shouldBlockIfFail), SemPISafeGive(&Sem)

Binary Semaphores:

SemBCreate (&Sem, initialHoldingCount, policy), SemBTake (&Sem, timeoutvalue, shouldBlockIfFail), SemBGive (&Sem)

Counting Semaphores:

SemCCreate (&Sem, initialHoldingCount, maximumAllowedHoldings, policy), SemCTake (&Sem, timeoutvalue, shouldBlockIfFail), SemCGive (&Sem)

Tasks:

TaskSpawn (&Task, priority),

Block-IO:

Block (time_units)

Interrupts:

DisableIntr (), EnableIntr ()

Busy : Hold_CPU_For (time_units)

Please notice that lower value for the priority indicates a higher priority.
It means that a task that has been spawned with priority 20 will get precedence over a task that has been spawned with priority 40.

Q1-

- (3)What is the difference between a binary semaphore and a regular counting semaphore?
    a. A binary semaphore can be owned by one task only. A regular counting semaphore can be owned by more than one task.
- (3)How would you initialize a binary semaphore so that the first task that tries to acquire it will become blocked?
    a. I would initialize it to 0 available tokens – empty state.
- (11)Here is a multitasking program that consists of a primary task t1 that is running at priority 20. t1 invokes t1_func. t1 spawns two additional tasks t2 and t3, which invoke t2_func and t3_func respectively. Please modify the program below in a way that CODE_FRAGMENT_T2_2 will be executed only after CODE_FRAGMENT_T3_T1 has completed execution and before CODE_FRAGMENT_T3_2 starts executing. You must not change priority of the tasks.

```
t1_func ()
{
  SemBCreate (&SemA,  …)
  taskSpawn(&t2, 40)
  taskSpawn(&t3, 50)
 }

t2_func()
{
CODE_FRAGMENT_T2_1
SemBTake (&SemA,  …, BLOCK_IF_FAIL);
CODE_FRAGMENT_T2_2
}

t3_func()
{
CODE_FRAGMENT_T3_1
SemBGive (&SemA)
CODE_FRAGMENT_T3_2
}
```

Q2-

- (8)Explain the difference between round-robin and priority-preemptive scheduling algorithms according to the sub-questions listed bellow:
    - What is the major requirement from tasks that are subject to participate in round-robin scheduling?
        - They should all share the same priority level.
    - How will the scheduler operate on these tasks if round-robin policy has been selected?
        - Each one of them will be given an equal time slice for running. The scheduler will transfer control to the next task one the current running task has reached its time slice.
    - How will the scheduler operate on these tasks if priority-preemptive policy has been selected?
        - Unless the running task will give up the CPU – other tasks with the same priority will not be able to use the CPU and the scheduler will not interfere.
    - Is there a difference between the behavior of a system that uses a round-robin policy and a system that uses a priority-preemptive policy – with respect to tasks that don't comply with the requirement imposed by 2.1.1?
        - No. Tasks with higher priority will always gain the CPU even on roind-robin system.
- (9)Here is a multitasking system that consists of a primary task t1 that is running at priority 20. t1 invokes t1_func. t1 spawns two additional tasks t2 and t3, which invoke t2_func and t3_func respectively. A global int variable myGlobal is to initialized to 15. What will its final value be according to the sub-questions listed bellow?
    - Using round-robin policy. - 2015
    - Using priority-preemptive policy. - 1015
    - T1 task has been initialized with the priority 60. - 1015

```
/*
BLOCK function puts t1 in a blocking state infintely.
*/
t1_func ()
{
 myGlobal = 15
 taskSpawn(&t2, 40)
 taskSpawn(&t3, 40)
 Block(INFINITE)
 }

t2_func()
{
 for (i =0; i < 1000; i ++) {
   Hold_CPU_For (1000)
   ATOMIC_INCREMENT(myGlobal)
  }
  Hold_CPU_For (FOREVER)
}
```

```
t3_func()
{
  for (i =0; i < 1000; i ++) {
    Hold_CPU_For (2000)
    ATOMIC_DECREMENT( myGlobal)
  }
  Hold_CPU_For (FOREVER)
}
```

Q3-

- (2)Explain what are the differences between a mutex and a binary semaphore.
  - A mutex can be released only by the task that owns it.
  - Binary semaphores can be initialized to empty state. Mutexs – are always initialized to available state (according to their definition).
- (2)How would you synchronize tasks that need to access some global resource?
  - I will use a Mutex.
- (2)Will you use this method to synchronize tasks and ISR access?
  - No. Tasks that fail to own the Mutex it should block – but ISR can't block.
- (11)The function insert_value is used in a multitasking system by some tasks to access a cyclic buffer and modify a specific cell in it. There is also a task in the system that is invoked periodically to release semaphores that hasn't been released by their owneres for long periods of time. As you can see from the code bellow, there is a serious potential of buffer-overrun while accessing the buffer. Please add code which prevents this overrun state. Please add code which prevents this overrun state. Try to use the most effective synchronization method. Hints:
  - You can use local variables in the function insert_value as C language permits and use it.
  - You can also change the order of the commands.

```
struct  MY_STRUCT {
  int val1;
  int val2;
}
global struct MY_STRUCT my_struct[100];
global int cyclic_idx = 0;
/* In the main task */
SemMCreate (&SemA, …)

insert_value (int value1, int value2)
{
  int local_index;

  SemMtake(&SemA,.., BLOCK_IF_FAIL);
  local_index = cyclic_idx ++;
  if (cyclic_idx == 100)
    cyclic_idx = 0;
  SemMGive(&SemA);

  my_struct[local_index].val1 =value1;
  my_struct[local_index].val2 =value2;

  delay_block(2);
}
```

Q4-

- The system referred to in this question is similar to the one described in Q3. However, ISRs may also call the function insert_value.
  - (3)Why can't we use mutex or semaphores for this issue?
    - ISR can't block on a mutex.
  - (3)Is it a good approach to disable interrupts while a task is blocked on an IO/timer event?
    - No. timer operation is based on the availability of interrupts.
  - (11)Please modify the code of insert_value in order to prevent buffer overrun.

```
insert_value (int value1, int value2)
{
  int local_index;

  DisableIntr ();
  local_index = cyclic_idx ++;
  if (cyclic_idx == 100)
    cyclic_idx = 0;
  EnableIntr ();

  my_struct[local_index].val1 =value1;
  my_struct[local_index].val2 =value2;

  delay_block(2);
}
```

Q5-

- (3)What is the "priority inversion" phenomena?
  - A task with low priority owns a resource while a high priority task is blocked on it, and an intermediate priority task is running.
- (3)Describe how priority inheritance algorithm provides a solution to this issue.
  - Once the higher priority task blocks, the mutex calls the scheduler in order to elevate the priority of the owning task to the maximum priority among all tasks that are blocked on it. Therefore, intermediate priority task will not interfere. Once the owner will release the resource – it will get its original priority back– and the higher priority task will preempt it and start its execution. Mutexes that provide this capability are using PRIORITY policy.
- (11)Here is a system that consists of 4 tasks: Task1, Task2, Task3 and Task4, which invoke T1_func – T4_Func respectively. Task1 is the first running task and its priority is 50.
  - Provide or draw a time-table, with brief comments of the events that cause re-scheduling.
  - Is task1 prevented from running by task4? Please explain.
    - § No. We can see that though task4 needs the cpu for infinite period of time – it never prevents t1 from running.

```
t1_func ()
{
SemMPISafeCreate (&SemA, POLICY_PRIORITY)
TaskSpawn (&T2, 100);
TaskSpawn (&T3, 150);
TaskSpawn (&T4, 200);
Block (15);
Hold_CPU_For (10)
}


t2_func ()
{
Hold_CPU_For (1)
Block (10)
SemMPISafeTake (&SemA, INFINITE_TIMEOUT, BLOCK_IF_FAIL)
Hold_CPU_For (20)
SemMPISafeGive (&SemA)
}

t3_func ()
{
Hold_CPU_For (1)
Block (5)
Hold_CPU_For (70)
}

t4_func ()
{
Hold_CPU_For (1)
SemMPISafeTake (&SemA, INFINITE_TIMEOUT, BLOCK_IF_FAIL)
Hold_CPU_For (20)
SemMPISafeGive (&SemA)
Hold_CPU_For (INFINITE)
}
```

Using a mutex that doesn't use priority inheritance.

Seq	Task	Running Time	Priority	Event	Mutex state	Total Time	Next transition-to_ready
1	T1	4	50	Block(15)	SemA=1	4	19
2	T2	1	100	Block(10)		5	15
3	T3	1	150	Block(5)		6	11
4	T4	5	200	Pre-by-t3	SemA=0	11	
5	T3	4	150	Pre-by-t2		15	
6	T2	1	100	Block_Mutex_A	SemA=-1	16	
7	T3	3	150	Pre-by-t1		19	
8	T1	10	50	Compl		29	
9	T3	63	150	Compl		92	
10	T4	18	200	Pre-by-t2 on SemRelease	SemA=0	110	
11	T2	21	100	Compl	SemA=1	131	
12	T4	FOREVER	200				

Using a mutex that uses priority inheritance.

Seq	Task	Running Time	Priority	Event	Mutex state	Total Time	Next transition-to_ready
1	T1	4	50	Block(15)	SemA=1	4	19
2	T2	1	100	Block(10)		5	15
3	T3	1	150	Block(5)		6	11
4	T4	5	200	Pre-by-t3	SemA=0	11	
5	T3	4	150	Pre-by-t2		15	
6	T2	1	100	Block_Mutex_A	SemA=-1	16	
7	T4	3	100	Pre-by-t1		19	
8	T1	10	50	Compl		29	
9	T4	15	100	Pre-by-t2 on Semrelease	SemA=0	44	
10	T2	21	100	Compl	SemA=1	65	
11	T3	66	150	Compl		131	
12	T4	FOREVER	200				

Q6-

Here is a system that consists of 5 tasks: Task1- Task5, which invoke T1_func to T5_func respectively. Task1 is the first running task and its priority is 50.

- (8) Semaphore policy is POLICY_FIFO. Provide or draw a time-table, with brief comments of the events that cause re-scheduling.
- (8)Semaphore policy is POLICY_PRIORITY. Provide or draw a time-table, with brief comments of the events that cause re-scheduling.
  - The time table will be similar to the one referring to POLICY_FIFO. The reason is that the first task that became blocked on the semaphore is t4 which has higher priority than t5.

```
Global SemA;
T1_func()
{
 SemCCreate (&SemA, 2, 10, policy)
 TaskSpawn (&T2, 60);
 TaskSpawn (&T3, 70);
 TaskSpawn (&T4, 80);
 TaskSpawn (&T5, 90);
  Hold_Cpu_For (3)
}
T2_func()
{
  SemCTake (&SemA, INFINITE, BLOCK_IF_FAIL);
 Use_Cpu_For (10);
 Block (40);
 Use_Cpu_For (1);
}
T3_func()
{
  SemCTake (&SemA, INFINITE, BLOCK_IF_FAIL);
 Use_Cpu_For (1);
 Block (20);
 SemCRelease(&SemA);
 Use_Cpu_For (1);
}
T4_func()
{
  SemCTake (&SemA, INFINITE, BLOCK_IF_FAIL);
 Use_Cpu_For (1);
 Block (20);
 SemCRelease(&SemA);
 Use_Cpu_For (1);
}
T5_func()
{
 Block (4);
 SemCTake (&SemA, INFINITE, BLOCK_IF_FAIL);
 Use_Cpu_For (1);
 Block (20);
 SemCRelease(&SemA);
 Use_Cpu_For (INFINITE);
}
```

Seq	Task	Running Time	Priority	Event	Semaphore state	Total Time	Next transition-to_ready
1	T1	8	50	Compl	SemA=2	x-start count from next raw	
2	T2	11	60	Block(40)	SemA=1	11	51
3	T3	2	70	Block(20)	SemA=0	13	33
4	T4	1	80	Block_sem_A	SemA=-1	14	
5	IDLE	4		Pre-by-t4		18	
6	T5	1	90	Block_sem_A	SemA=-2	19	
7	IDLE	14		Pre-by-t3		33	
8	T3	2	70	Compl	SemA=-1	35	
9	T4	1	80	Block(20)		36	56
10	IDLE	15		Pre-by-t2		51	
11	T2	1	60	Compl		52	
12	IDLE	4		Pre-by-t4		56	
13	T4	2	80	Compl	SemA=0	58	
14	T5	FOREVER	90		SemA=1		