IMPERIAL COLLEGE LONDON

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
EXAMINATIONS 2003

OPERATING SYSTEMS

Tuesday, 3 June 2:00 pm

Time allowed: 2:00 hours

Corrected Copy

There are FOUR questions on this paper.

Q1 is compulsory.
Answer Q1 and any two of questions 2-4.

Q1 carries 40% of the marks. Questions 2 to 4 carry equal marks.

Any special instructions for invigilators and information for candidates are on page 1.

Examiners responsible First Marker(s) : T.J.W. Clarke
Second Marker(s) : G.A. Constantinides

Special Information for Invigilators

Each candidate must be given the booklet entitled uC/OS-II Reference.

Information for Candidates

Information about uC/OS-II can be found in booklet uC/OS-II Reference.

The Questions

1. (a) Binary semaphores, and global interrupt disabling, are two methods that can be used in the uC/OS-II RTOS to implement mutually exclusive access to a shared resource. Describe briefly these methods, writing pseudocode that uses the appropriate system calls and/or macros. Compare the effects on system performance and latency of the two methods.

[6]

(b) Explain the meaning of bilateral rendezvous. Write pseudocode using uC/OS-II system calls to implement bilateral rendezvous between two tasks, using semaphores.

[5]

(c) Interrupt response time and task-level response time are two important determinants of RTOS performance. Which is longer? Contrast the merits of running time-critical code at interrupt and task level.

[6]

(d) In uC/OS-II task scheduling is implemented by OS_Sched(). Describe precisely the effect on tasks of calling this function, and under what circumstances task pre-emption will happen. Your answer should distinguish clearly between ready-to-run, running, and waiting tasks.

[6]

(e) The uC/OS-II RTOS uses a tick interrupt generated by hardware at regular intervals. Explain how this is used, detailing the system-level actions that occur as the result of this interrupt. Discuss whether task pre-emption would be possible without the tick interrupt?

[6]

(f) In the uC/OS-II RTOS semaphores, and event flag groups, use different implementations for the list of waiting tasks. Compare and contrast the two implementations, explaining in each case why they were chosen.

[6]

(g) An application is currently running under uC/OS-II configured with semaphore, message box and message queue services enabled. On inspection of the code it is noted that all semaphores used are configured as binary semaphores. Suggest a possible optimisation to the uC/OS-II configuration that would save code space whilst keeping functionality.

[5]

2. (a) Explain the use of the task ready list in uC/OS-II, specifying precisely the conditions for a task to be in the task ready list.

[5]

(b) Describe the implementation of the task ready list in uC/OS-II, illustrating your answer by showing the contents of OSRdyTbl and OSRdyGrp when the task ready list initially contains tasks 3,4,11,15, and explaining how OSUnMapTbl is used to determine the highest priority ready task.

[10]

(c) In uC/OS-II it is never possible for the ready task list to be empty, so functions such as OS_Sched() which calculate the highest priority ready task will never fail. Explain how uC/OS-II ensures that an empty ready task list can never happen.

[5]

(d) In a modified version of uC/OS-II there are maximum of 1023 tasks, numbered 1-1023. Propose new data structures for OSRdyTbl, OSRdyGrp, OSMapTbl, OSUnMapTbl which will implement the ready list in this system, quantifying the necessary increase in memory resources.

[10]

3. (a) What are the advantages and disadvantages of foreground/background operating systems when compared with pre-emptive multi-tasking operating systems?

[10]

(b) A foreground/background system runs on a CPU with 16 prioritised interrupt levels (15 = lowest priority, 0 = highest priority). It has three time-critical jobs *J1*, *J2*, *J3* which occur repetitively at periods of 10us, 21us and 71us respectively. The execution time of each job is *e1*, *e2* and *e3* respectively. Each job has a hard deadline equal to its period (thus each computation must finish executing before the next computation starts). Explain how you would assign jobs to interrupt priority levels so that the likelihood of hard deadlines being met is maximised, and state the rate monotonic scheduling condition on *e1*,*e2*,*e3* for all hard deadlines to be met (interrupt switch time may be ignored).

[10]

(c) Three jobs *J1*, *J2*, *J3* are synchronised with periods of 10us, 20us, 40us respectively, CPU utilisation 50%, 25%, 25% respectively, and hard deadlines equal to their periods. You may assume that all jobs become ready to run simultaneously at $t = 0$. Show by drawing an appropriate timing diagram illustrating when each job executes how 100% CPU utilisation can be achieved whilst meeting all hard deadlines. Comment on the application of the rate monotonic scheduling utilisation limit in this case.

[10]

4. (a) Two tasks, *A* and *B*, both use a binary semaphore S to share a mutually exclusive resource. *A* has higher priority than *B*. Explain, giving the sequence of task switches and semaphore operations on S, how priority inversion can cause execution of *A* to be blocked by a task *C* of lower priority than *A*. What is the condition on the priority of *C* necessary for this to happen?

[10]

(b) *Figure 4.1* details the maximum CPU run length before waiting, and required task-level response time, of 3 tasks *X*, *Y* and *Z*. You may assume that all tasks wait for a time much greater than these run lengths, and that OS task-level response time is 15us. Calculate the worst-case task-level response times to each task given other tasks priorities, and hence determine all permitted priority orders of the three tasks.

[10]

(c) Describe how the priority inversion in (a) can be avoided under uC/OS-II by using a mutex semaphore, explaining carefully how dynamic task priority is implemented by the mutex system code.

[10]

Task	**Maximum run-length**	**Required response-time**
X	*50us*	*50us*
Y	*30us*	*100us*
Z	*200us*	*95us*

Figure 4.1

Solution to Question 1

a)

Interrupt switching, bracket critical section in code that switches interrupts on & off

```
OS_ENTER_CRITICAL();
/* critical section */
OS_EXIT_CRITICAL();
```

(binary) semaphores. Execute critical section only when key of binary semaphore is obtained, release it at end.

```
/* initialise semaphore with count=1 */
OSSemPend()
/* critical section */
OSSemPost()
```

Switching interrupts is very fast, but will increase interrupt and task level response time if critical section is longer than any other section with interrupts switched off. Semaphores are slower, and use system resources for the semaphore structure, but allow arbitrarily long critical sections.

b)

Bilateral rendezvous is when two tasks each wait for the other to reach a designated point before both can then proceed. Use two binary semaphores, one for each task:

Initialise both semaphores with count=1.

Task A:

OSSemPost(SemB) /* signal task B have got here */

OSSemPend(SemA) /* wait for task B */

Task B:

OSSemPost(SemA)

OSSemPend(SemB)

c)

Task-level response is always longer than interrupt-level response. Code in interrupt runs with smaller latency (interrupt level instead of task-level) but can't make use of OS task-level communication primitives. Also interrupt code increases worst case latency of all task-level code - this is usually unacceptable unless code is very short. A task, on the other hand, can be given a priority appropriate to its function with faster operations scheduled at a higher priority and unaffected.

d)

OS_Sched() normally examines the list of ready tasks and schedules the highest priority task. If this is the current (running) task there is no change, otherwise a ready-to-run task is run, and the current task, ir running, changes state to reday-to-run. No change is made if either scheduling is locked, or OS_Sched is called from a nested interrupt level (in the latter case OS_Sched is guaranteed to be called on exit from the nested interrupts).

Task pre-emption occurs if there is a ready-to-run task with higher priority than the current task, and the current task is also ready-to-run.

e) The tick interrupt is used to increment a global counter (OSTime), and check the TCB of every system task. If .OSTCBDLY is non-zero the task is waiting on a delay and this value must be decremented. If the resulting value is zero the task must be checked to see if it is waiting on anything else, and if not woken up. Finally, after all tasks have been so checked OS_Sched is called which will if necessary pre-empt the current task with a higher priority task.

f) Semaphores use a bit-map implementation of the ready list which is very efficient in both space and, more critically time. Event Flag Groups use a doubly-linked list implementation which is less efficient in both space and time. However it has the important advantage that the list of currently waiting tasks can be accessed (by chaining down the list). This operation is not easy to implement in the bit-mapped list. It is required since when a flag is changed each task must be checked to see whether its wakeup condition is now met. Also, the linked list provides a place to store the wakeup condition information.

g) The functionality of binary semaphores can be provided using message boxes, so the semaphore code could be removed and application semaphores rewritten using message boxes.

Solution to Question 2

a) The task ready list represents the set of tasks currently ready to run. A task is in the list if it is either ready to run or currently running.

b)

OSRdyTbl is a bit-map with bits 1 for ready tasks. Bits are ordered by priority, since each task has a unique priority.

OSRdyGrp contains 1 bit for each byte in OSRdyTbl: this bit is 1 if any of the corresponding OSRdyTbl bits is non-zero. This allows quick determination of the highest priority task.

OSRdyGrp

						1	1

OSRdyTbl

			1	1			
1				1			

(all other bits 0).

OSUnMapTbl is a 256 byte table.

The value of OSUnMapTbl[x] is the bit index of the lowest index 1 in x. Therefore it is used to determine first the index of the lowest 1 in OSRdyGrp, say y. Then it is used again to determine the lowest index 1 in OSRdyTbl[y], which corresponds to the lowest number (highest priority) task in the ready list.

c)

The idle task is defined with the lowest possible priority, and always runnable. It runs when no other task can run, and, as a useful side-effect, is used to calculate the total system idle time, by continuously incrementing a counter. Since systems are defined always to have the idle task, the ready list is never empty.

d)

Increase OSRdyTbl to 10 bits, and OSRdyGrp to 128 bytes. OSMapTbl must now have 10 elements, and a 16 bit result, of which only the bottom byte is used for OSRdyGrp bit manipulation. OSUnMapTbl needs to be 1024 bytes, and returns a number between 0 and 9. If this cost in space is unacceptable then a slower implementation could be used in which the top two bits are explicitly checked, rather than using a table lookup.

Solution to Question 3

a)

Foreground/background systems require no OS kernel code, since all code is run from interrupts or a single base-level loop. Thus they are simple and efficient. Switching between activities is performed by interrupts and is very fast when compared with task-level switching. However there is no simple way to implement threads of computation that can wait and synchronise with each other. Although task-based systems can be translated into the equivalent systems using interrupts (via event./action diagrams) the resulting code is very difficult to read and maintain. Therefore whenever application level code is required in practice a pre-emptive multi-tasking system is preferable.

b)

1) assign lower period jobs higher priority interrupts (only the order of the priorities matters, not the precise values). E.g.: J1=P0, J2=P1, J3=P2. Given this, the RMS condition is that if total CPU utilisation is less than:

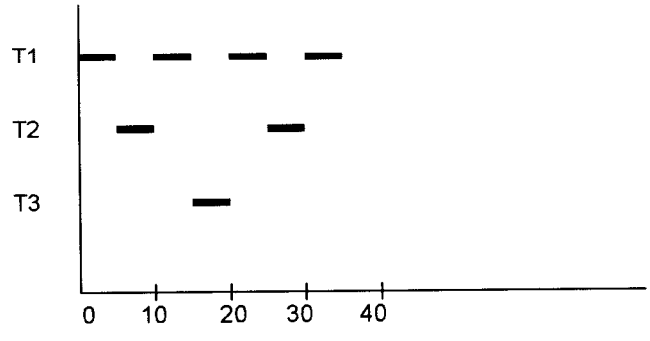
$3(2^{1/3} - 1) = 0.780$. Therefore:

$e1/10+e2/21+e3/71 < 0.78$ => hard deadlines always met.

c)

Set higher priorities to lower periods. Tasks are:

Task	Period	Length
T1	10	5
T2	20	5
T3	40	10



In this case, because the times at which jobs occur are synchronised, higher utilisation than that guaranteed by RMS theorem is possible. Note that the RMS theorem is a guaranteed lower limit, below which all systems will meet deadlines, it does not imply that higher utilisations are impossible.

Solution to Question 4

a)

Suppose C has priority greater than B but less than A. If C preempts B while B holds semaphore key, and then A waits on key, A must wait until after C waits, even though it has higher priority than C.

B pends on S

C preempts B

A preempts C

A pends on S, and waits. C is scheduled

C runs till it next waits. Then B runs

B posts to S

A is woken up, and pre-empts B

b)

Task X. Can have no tasks or Y greater priority (response-time 15 or 45us)

Task Y. Can have no tasks or X greater priority(response time 15 or 65us)

Task Z. Can have X and Y, or X or Y greater priority. (response time 95us or less)

Thus allow X > Y > Z or Y > X > Z

c)

A mutex semaphore boosts the priority of a task X that has acquired its key to a fixed higher priority (defined at task creation time) which must not be used by any other task. The priority boosting occurs when a higher priority task Y pends on the mutex until X posts to the mutex. If the higher priority is greater than that of all possible tasks pending on the mutex, priority inversion is avoided.