

Question #1 (Basic Concepts)

Briefly answer each of the following questions in the space provided.

- (a) What is meant by a “critical section”? What regions in a system’s address space must be treated as critical sections? Give one simple example for each of (1) a critical section involving a data structure, and (2) a critical section involving a hardware register. What would be some common ways of correctly handling critical sections of types (1) and (2)?

[15 marks]

A critical section is a segment of code that must be executed until completion, without being interrupted by any context switches. For certain critical sections, not even hardware interrupts are allowed to run.

Examples of critical sections include the following:

- (1) A routine that updates a data structure (e.g. a linked list, or a binary tree) consisting of records that are linked together by pointers typically has critical sections if two or more pointers must be changed when the data structure is changed from one consistent state to the next consistent state.
- (2) A hardware register (e.g. the interrupt status register in the DUART) may need to be read and then written in a certain sequence of operations before any other program can be allowed use the hardware. Such a situation commonly occurs during hardware initialization routines or interrupt service routines.

The first kind of critical section could probably be handled by simply disabling context switches. However, if interrupt service routines could potentially need to access the data structure in question, then those interrupts would need to be masked out as well.

The second kind of critical section could also be handled by disabling context switches and interrupts. However, depending on the way the interrupts are prioritized, it may be sufficient to rely on the interrupt priority systems to protect the critical section.

Question #1 (Basic Concepts, cont'd)

- (b) In the Internet, users want to take advantage of applications that need to have reliable network connections that carry streams of ordered bytes in both directions. Unfortunately, the available physical networks are a mixture of various types of data packet networking technologies (e.g. coaxial cable, twisted pair, optical fibre, wireless), which differ in (1) their bit rate, (2) their reliability, (3) their packet structure and packet lengths, and (4) the protocols used to transport data packets from node to node. Briefly describe how the Internetworking Protocol (IP) and the Transmission Control Protocol (TCP) co-operate to overcome the technical challenges posed by points (1), (2), (3) and (4), and provide the reliable connections that applications need to have.

[15 marks]

- (1) Differences in bit rate are accommodated by standard data buffering and flow control mechanisms. TCP uses a window-based flow control mechanism that ensures that all transmitted bytes are acknowledged by the far end.
- (2) Differences in reliability are also accommodated by the TCP byte acknowledgement mechanism, which ensures that the loss of data bytes is detected and corrected by packet retransmission. That being said, TCP/IP is basically an optimistic protocol that assumes that most connections will be quite reliable. This optimism is evident in the fact that the checksums that protect the data payloads against errors are only evaluated once at the destination node, and not at each intervening tandem node.
- (3) Differences in the packet structure of the protocols below TCP/IP are accommodated by allowing IP datagrams to be broken up or combined, as necessary, to fit into the available payload size of the physical layer. Protocols above TCP/IP are accommodated by supporting, as the dataload, any arbitrary ordered sequence of bytes.
- (4) Differences in node-to-node protocols are accommodated by using IP conventions (e.g. IP addressing) throughout the network. IP datagrams do not have to be transmitted reliably from node-to-node, and they can be transported from node to node by just about any packet-based physical layer protocol.

Question #2 (MicroC/OS-II)

- (a) MicroC/OS-II uses strictly ordered priorities to ensure that it is always clear which of the one or more ready-to-run tasks should be running on the CPU. Given that a highest priority task is currently running on the CPU, how is it possible for other lower priority tasks to ever get a chance to run on the CPU? In your answer, be sure to mention which software-triggered and interrupt-triggered events can cause a context switch that can stop the highest priority task from running on the CPU.

[15 marks]

The highest priority ready-to-run task is always awarded the right to execute on the CPU. They can relinquish the CPU as a result of software-triggered events by either (1) suspending or terminating themselves, (2) decreasing their own priority so that it loses its position as highest priority ready-to-run task; or (3) blocking on some event (e.g. a message queue or semaphore) that prevents their further execution.

Interrupt handling (4) is another way in which code can be executed despite the status of the highest-priority ready-to-run task. The interrupt service routine (ISR) could cause a second task to be unblocked (say by signaling a binary semaphore) and this second newly ready-to-run task could have a higher priority than the currently running task. In such a scenario a context switch would be triggered at the end of the ISR.

Question #2 (MicroC/OS-II, cont'd)

- (b) Briefly describe how an application programmer gets a MicroC/OS-II multitasking environment up and running. Be sure to describe all of the function calls and variable assignments in the initialization steps that occur at the C program level. Why do you suppose some actions are performed in a main program while others are performed in a special task?

[15 marks]

Before the "begin()" routine:

- (1) The MicroC/OS-II library header file must be included.
- (2) Space must be allocated for the stacks of all tasks.
- (3) Pointer variables for OSEVENTs must be declared.
- (4) Function prototypes for all task functions must be declared.

Within the begin() routine:

- (1) The exception vector must be initialized for the context switch routine.
- (2) The function OSInit() must be called to initialize data structures within MicroC/OS-II.
- (3) OSTaskCreate() must be called once to create the start-up task.
- (4) As the very last step of the begin() routine, OSStart() must be called to start multitasking.

Within the start-up task:

- (1) Function TickInit() must be called to initialize the hardware timer.
- (2) If the Statistics Task has been enabled in the configuration file, then OSStatInit() must be called.
- (3) OSTaskCreate() must be called to create each of the application tasks.
- (4) As its last action, the start-up task should call OSTaskDelete() to delete itself.

Some initialization steps (e.g. space declaration for stacks, declaration of pointers to OSEVENTs, context switch interrupt vector loading) are required prior to the start of multitasking. Other steps (e.g. application task creation) are best performed after multitasking is running so that all of the regular MicroC/OS-II functions can be used. So it is convenient to spread initialization over both a main program and a special start-up task.

Question #3 (Assembly Language Programming)

Write an assembly language subroutine, called REFORMAT, that receives one 32-bit *pointer*, one 16-bit unsigned *startcount* value, and one 16-bit unsigned *stopcount* value. The three parameters are to be passed on the stack so that immediately before REFORMAT is called, the pointer is pushed onto the stack first, followed by startcount, followed by stopcount. The pointer points to the first byte in the first long word in an array of packed long words. The longwords are numbered starting at zero for the long word pointed to by the first parameter. Your routine is to scan through the array of long words and to rotate right by *three* bit positions those long words at those positions numbered from startcount to stopcount, inclusive. You do not have to check the validity of the input parameters.

[40 marks]

```
REFORMAT:  MOVEM.L  A0/D0-D3,-(SP)  // save CPU regs

           MOVE.L   28(SP),A0      // retrieve array pointer
           MOVE.W   26(SP),D0      // retrieve start count
           MOVE.W   24(SP),D1      // retrieve stop count

           CLR.L    D2             // create offset to first longword
           MOVE.W   D0,D2
           LSL.L    #2,D2          // times 4 for byte offset

REF_LOOP:  MOVE.L   (0,A0,D2.L),D3  // read next longword
           ROR.L    #3,D3          // rotate the longword
           ROR.L    D3,(0,A0,D2.L) // write the longword

           ADDQ.L   #4,D2          // update byte offset
           ADDQ.W   #1,D0          // update longword counter
           CMP.W    D1,D0          // check for last longword
           BLS     REF_LOOP

           MOVEM.L  (SP)+,A0/D0-D3 // restore CPU regs
           RTS
```