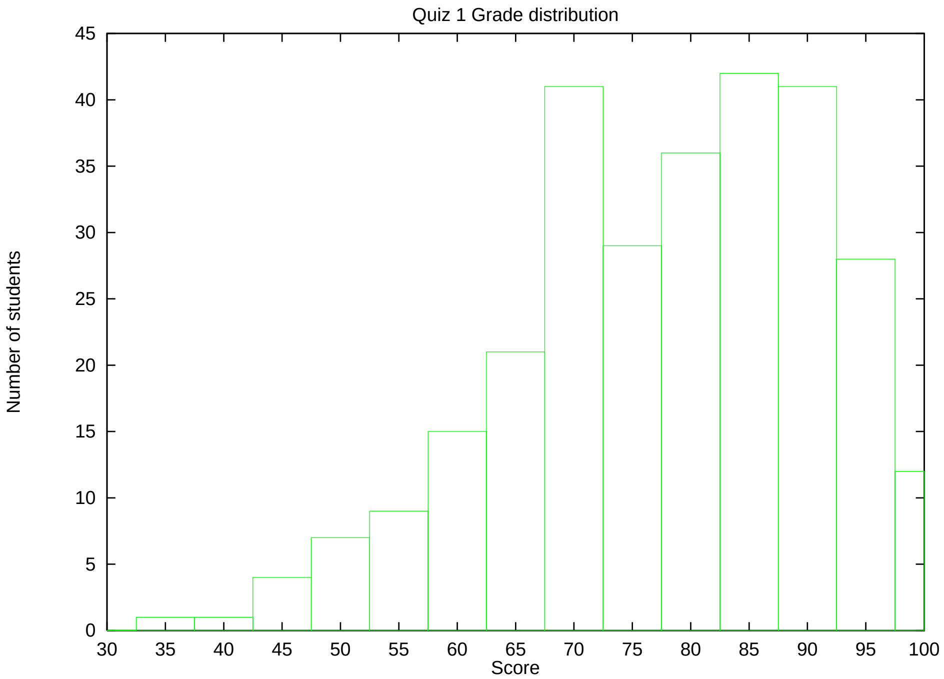
*Department of Electrical Engineering and Computer Science*

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

**6.033 Computer Systems Engineering: Spring 2000**

# Handout 15 - Quiz I Solutions

Average: 75
Median: 77
Std. Dev: 13




**Name: Solution**

# I Reading questions

**1. [6 points]:** Lucky reports that according to N. Negroponte 16-year olds are his main competition for innovation. Why does Lucky agree?

**(Circle ALL that apply)**

**A.** The Media lab doesn't teach anything useful.

**B.** Professional experts cannot absorb change at the rate that current technology is changing.

**C.** Predicting the future in a time with many changes doesn't work.

**D.** Non-experts have as good a chance as anyone of coming up with the next idea.

**2. [6 points]:** What problems contributed to the disasters (i.e., multiple people dying) described in the paper about the Therac-25?

**(Circle ALL that apply)**

**A.** Lack of any hardware mechanism to prevent beam turn-on if the turntable isn't in position.

**B.** Difficulty to reproduce problems, because errors are timing dependent.

**C.** A software bug in which a variable overflows.

**D.** Failure to follow through on initial problem reports.

**3. [6 points]:** Why does the X window system use streams instead of RPCs?

**(Circle best answer)**

**A.** RPC's roundtrip-delay across the network would hurt throughput.

**B.** Programming with RPC is more difficult than programming with streams.

**C.** RPC is unreliable.

**D.** Streams are a better mechanism for fault isolation.

**4. [6 points]:** Ethernet

**(Circle ALL that apply)**

**A.** Delegates arbitration of conflicting transmissions to each station.

**B.** Always guarantees the delivery of packets.

**C.** Can support unbounded number of computers.

**D.** Has limited physical range.

*Although the paper by Metcalfe and Boggs points out that gateways would permit unbounded physical range and address cover, the result is not an Ethernet, it is a network consisting of several Ethernets. Each individual Ethernet remains limited in both address cover and physical range.*

**5. [6 points]:** What are potential advantages of an object-based virtual memory as described in the OOZE paper versus a page-based virtual memory system as described in the class notes?

**(Circle ALL that apply)**

$\boxed{\textbf{A.}}$ Better utilization of memory, because objects are more likely than pages to contain data that applications are going to use.

$\boxed{\textbf{B.}}$ Better locality, because the parts of an object belong together.

**C.** Better disk performance when writing an object back to the storage device, because objects are variable sized.

**D.** Larger page tables, because objects are smaller than pages.

*Although it might seem that writing objects that are smaller than a disk sector would improve disk performance, it just leaves blank spaces on the disk that the read/write heads still have to pass over, so overall disk performance is unchanged.*

**6. [6 points]:** When does Eraser report false positives?

**(Circle ALL that apply)**

$\boxed{\textbf{A.}}$ When analyzing a program with private locks.

$\boxed{\textbf{B.}}$ When a program is carefully designed to access shared data in phases, and when some of these phases access shared data read-only, without synchronization.

**C.** When reporting errors in code that doesn't execute.

**D.** When the lock set algorithm computes a non-empty set.

# II A Picokernel-based Stock-ticker system

After the recent launch of Windows 2000, Ben Bitdiddle decides to design a computer system based on a new kernel architecture, picokernels, and a new hardware platform, simplePC. Ben has paid attention to the first lectures in 6.033 and is going for extreme simplicity. The simplePC platform contains one simple processor, a virtual memory manager (which translates the addresses issued by the processor), a memory module, and an input and output device. The processor has two special registers, a program counter (PC) and a stack pointer (SP). The SP points to the value most recently pushed on the stack.

The calling convention for the simplePC processor uses a simple stack model:

- A call to a function pushes the address of the instruction after the call onto the stack and then jumps to the function.
- Return from a function, either through the explicit use of `return`, or when the function finishes execution, pops the address from the top of the stack and jumps to it.

Programs on the simplePC don't use local variables. Arguments to functions are passed in registers, which are **not** saved and restored automatically. Therefore, the only values on the stack are return addresses.

Ben develops a simple stock-ticker system to track the stocks of the startup he joined. The program reads a message containing a single integer from the input device and prints it on the output device:

```
100. int msg;
101. boolean input_available;

1. read_input() {
2.   while (TRUE) {
3.     while (input_available == FALSE) ;  // idle loop
4.     print_msg(msg);
5.     input_available = FALSE;
6.   }
7. }

200. boolean output_done;
201. int *OUTPUT_ADDRESS;       // pointer to where output should be stored

12. print_msg(int m) {
13.   *OUTPUT_ADDRESS = m;            // stores m into the location pointed
                                      // to by OUTPUT_ADDRESS
14.   while (output_done == FALSE);   // idle loop
15.   output_done = FALSE;
16. }

17. main() {
18.   read_input();
19.   halt;                           // shutdown computer
20. }
```

The program contains two functions: `read_input` and `print_msg`. `read_input` spin-waits until `input_available` is set to `TRUE` by the input device (the stock reader). When the input device receives a stock quote, it places the quote value into `msg` and sets `input_available` to `TRUE`.

`print_msg` prints the message on an output device (a terminal in this case); it writes the value stored in the message to the device and waits until it is printed; the output device sets `output_done` to `TRUE` when it finishes printing.

The numbers on each line correspond to addresses as issued by the processor to read and write instructions and data. You may assume that each line of the pseudo-code can be compiled into one machine instruction.

**7. [6 points]:** These numbers are?

**(Circle best answer)**

**A.** virtual addresses.

**B.** physical addresses.

**C.** page numbers.

**D.** offsets in a virtual page.

Ben runs the program directly on simplePC, starting in `main`, and at some point he observes the following values on the stack (remember, only the stock-ticker program is running):

```
stack
  19
   5    <--- stack pointer
```

**8. [6 points]:** What is the value 5 on the stack?

**(Circle best answer)**

**A.** the return address for the next return instruction.

**B.** the return address for the last return instruction.

**C.** the value of PC.

**D.** the value of SP.

**9. [6 points]:** Which function is being executed by the processor?

**(Circle best answer)**

**A.** `read_input`

**B.** `print_msg`

**C.** `main`

**D.** none of the above.

**10. [6 points]:** `print_msg` writes a value to address `OUTPUT_ADDRESS`, which will end up on the terminal. What technique is used to make this work?

**(Circle best answer)**

**A.** memory-mapped I/O.

**B.** sequential I/O.

**C.** streams.

**D.** remote procedure call.

Ben wants to run multiple instances of his stock-ticker program on the simplePC platform so that he can obtain more frequent updates to track more accurately his current net value. Ben buys another input and output device for the system, hooks them up, and he implements a trivial thread manager:

```
300. int threadtable[2];            // stores stack pointers of threads.
                                    // arrays start with index at 0 (i.e.
                                    // first slot is threadtable[0])
302. int current_thread = 0;

21. yield() {
22.   threadtable[current_thread] = SP; // move value of SP into table
23.   current_thread = (current_thread + 1) % 2;
24.   SP = threadtable[current_thread]; // load value from table into SP
25.   return;
26. }
```

Each thread reads from and writes to **its own** device. Do not worry about allocation and management of stack frames. Ben also modifies `read_input` from Page 4:

```
100. int msg[2];                                  // CHANGED to use array
102. boolean input_available[2];                  // CHANGED to use array

30. read_input() {
31.    while (TRUE) {
32.       while (input_available[current_thread] == FALSE) {    // CHANGED
33.          yield ();                                          // CHANGED
34.       }                                                     // CHANGED
35.       print_msg(msg[current_thread]);         // CHANGED to use array
36.       input_available[current_thread] = FALSE; // CHANGED to use array
37.     }
38. }
```

Ben powers up the simplePC platform and each thread starts running in `main`. The two threads switch back and forth correctly. Ben stops the program temporarily and observes the following stacks:

```
stack of thread 0                      stack of thread 1
19                                     19
36          <-- stack pointer          34
```

**11. [6 points]:** Thread 0 was running (i.e. `current_thread == 0`). Where will thread 1 resume computation after thread 0 executes the return instruction in yield the next time?

**(Circle best answer)**

**A.** } (line 34).

**B.** `halt` (line 19).

**C.** `print_msg(msg[current_thread])` (line 35).

**D.** `input_available[current_thread] = FALSE;` (line 36).

**12. [6 points]:** What values can be on the stack of each thread?

**(Circle best answer)**

**A.** Addresses of any instruction.

**B.** Addresses where functions return to (e.g., 36).

**C.** Addresses of any data location.

**D.** Addresses of instructions and data locations.

Ben observes that each thread in the stock-ticker program spends most of its time polling its input variable. He introduces an explicit function that the devices can use to notify the threads. He also rearranges the code for modularity:

```
int state[2];

schedule_and_dispatch() {
  threadtable[current_thread] = SP;
  while (???) {
    current_thread = (current_thread + 1) % 2;
  }
  SP = threadtable[current_thread];
  return;
}

yield() {
  state[current_thread] = WAIT;
  schedule_and_dispatch();
}

notify(int n) {
  state[n] = RUNNABLE;
}
```

When the input device receives a new stock quote, the device interrupts the processor **immediately** and saves the PC of the currently running thread on the currently running thread's stack. Then, the processor runs the interrupt function. When the interrupt function returns, it pops the return address from the current stack, returning control to a thread. The pseudo code for the interrupt function is:

```
device(int n) {                          // interrupt for input device n
    push current thread's PC on stack pointed to by SP;
    while (input_available[n] == TRUE);  // wait until read_input is done
                                         // with the last input
    stores stock quote into msg[n];
    input_available[n] = TRUE;
    notify(n);                           // notify thread n
    return;                              // i.e., pop PC
}
```

During the execution of an interrupt function, interrupts are disabled. Thus, an interrupt function and functions that it calls (e.g., `notify`) are not interrupted. Interrupts are re-enabled when `device` returns.

Using the thread manager on Page 9, answer the following questions:

**13. [6 points]:** What expression should be evaluated on line 42 to ensure correct operation of the thread package?

**(Circle best answer)**

**A.** `state[current_thread] == WAIT`

**B.** `state[current_thread] == RUNNABLE`

**C.** `threadtable[current_thread] == SP`

**D.** `FALSE`

**14. [7 points]:** Assume thread 0 is running and thread 1 is not running (i.e., it has called `yield`). What event(s) needs to be happen before thread 1 will run?

**(Circle the smallest number of choices)**

**A.** Thread 0 calls `yield`.

**B.** The interrupt function for input device 1 calls `notify`.

**C.** The interrupt function for input device 0 calls `notify`.

**D.** No events are necessary.

**15. [7 points]:** What values can be on the stack of each thread?

**(Circle best answer)**

**A.** Addresses of any instruction, but not counting the device driver interrupt function.

**B.** Addresses of all instructions, including all device driver interrupt function instructions.

**C.** Addresses where functions return to (e.g., 36).

**D.** Addresses of instructions and data locations.

*Note that the address of the* `return` *instruction in the device driver can also be on the stack.*

**16. [8 points]:** Under which scenario can thread 0 deadlock?

**(Circle ALL that apply)**

**A.** When device 0 interrupts thread 0 just before the first instruction of `yield`.

**B.** When device 0 interrupts just after thread 0 completed the first instruction of `yield`.

**C.** When device 0 interrupts thread 0 between instructions 35 and 36 in the `read_input` function on Page 7.

**D.** When device 0 interrupts when the processor is executing `schedule_and_dispatch` and thread 0 is the WAITING state.