Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.033 Computer Systems Engineering: Spring 2003

Handout 13 - Quiz I

All problems on this quiz are multiple-choice questions. In order to receive credit you must fill in the blank(s) or mark the correct answer or answers for each question. You have 50 minutes to answer this quiz.

Write your name on this cover sheet AND at the bottom of each page of this booklet.

Some questions may be much harder than others. Read them all through first and attack them in the order that allows you to make the most progress. If you find a question ambiguous, be sure to write down any assumptions you make. Be neat. If we can't understand your answer, we can't give you credit!

THIS IS AN OPEN BOOK, OPEN NOTES QUIZ.
NO PHONES, NO LAPTOP, NO PDAS, ETC.

CIRCLE your recitation section number:

10:00	1. Lynch/Muthitacharoen	10. Horn/Bodhi	
11:00	2. Lynch/Li	6. Nagpal/Bodhi	11. Horn/Muthitacharoen
12:00	5. Nagpal/Li		
1:00	3. Jackson/Krevat	7. Saltzer/Dabek	8. Rinard/Laas
2:00	4. Jackson/Dabek	9. Saltzer/Laas	12. Rinard/Krevat

Do not write in the boxes below

1-4 (xx/30)	5-9 (xx/30)	10-13 (xx/40)	Total (xx/100)

Name:

I Reading questions

1. [8 points]: Which of the following problems contributed to the Therac-25 accidents according to Reading #4 ("An investigation of the Therac-25 accidents")?

(Circle ALL that apply)

A. The Therac-25 didn't have a safe fallback for failures.

B. New hardware interlocks prevented the beam from turning off.

C. Concurrency made anomalous behaviors hard to reproduce.

D. Only enforced modularity was used in the design.

2. [8 points]: Which of the following was highlighted in the Ritchie and Thompson paper about UNIX (reading #6, "The UNIX time-sharing system")?

(Circle ALL that apply)

A. The benefits of PC losering.

B. The advantages of a huge address space offered by virtual memory.

C. The value of uniform abstractions for devices.

D. The data structure design underlying the file system.

3. [8 points]: Which of the following considerations were important in the design of the Flash web server according to reading #7 ("Flash: an efficient and portable Web server")?

(Circle ALL that apply)

A. Network capacity exceeds maximum disk throughput.

B. UNIX does not provide an adequate asynchronous interface for disk I/O.

C. All web pages are small.

D. The entire content of a Web site may not fit in cache.

4. [6 points]: Ethernet, as described by reading #7 ("Ethernet: distributed packet switching for local computer networks"),

(Circle ALL that apply)

A. delegates arbitration of conflicting transmissions to each station;

B. always guarantees the delivery of packets;

C. can support an unbounded number of computers;

D. has limited physical range.

II Enforced Modularity and coordination

Ben develops an operating system for a simple computer. The operating system has a kernel that provides address spaces, threads, and output to a console.

Each application has its own user-level address space and uses one thread. The kernel program runs in the kernel address space, but doesn't have its own thread. (The kernel program is described in more detail below).

The computer has one processor, a memory, a timer chip (which will be introduced later in the problem), a console device, and a bus connecting the devices. The processor has a user/kernel mode bit. The processor has two sets of program counter (PC), stackpointer (SP), and page-map address registers (PMAR). A PMAR points to a page map that translates virtual addresses into physical addresses. One set for user space (kernel/user mode bit is set to user): `upc`, `usp`, and `upmar`. One set for kernel space (user/kernel bit is set to kernel): `kpc`, `ksp`, and `kpmar`. Only programs in kernel mode are allowed to store to `upmar`, `kpc`, `ksp`, and `kpmar`—storing a value in these registers is an illegal instruction in user mode.

The processor switches from user to kernel mode when one of three events happen: an application issues an illegal instruction, an application issues a supervisor call instruction (with the "svc" instruction), or the processor receives an interrupt in user mode. The processor switches from user to kernel mode using an atomic kernel-entering mechanism: it sets `kpc` to a default entry point in the kernel address space, it sets `ksp` to the top of a default stack in the kernel address space, and it sets the mode bit to kernel, causing the processor to use `kpc`, `kpmar` and `ksp`. (The user program counter, stack pointer, and page map address values stay in `upc`, `usp`, and `upmar`, respectively.)

To return from kernel to user space, a kernel program executes the `rti` instruction, which sets the kernel/user mode bit to user, causing the processor to use `usp`, `upc`, and `upmar`.

In addition to these registers, the processor has four general-purpose registers: `ur0`, `ur1`, `kr0`, and `kr1`. The `ur0` and `ur1` set is for user mode. The `kr0` and `kr1` set is for kernel mode.

Ben runs two user applications. Each executes the following program:

```
int t = 1;

void main () {
  while (1) {
    t = t + t;
    print(t);
    yield ();
  }
}
```

`print` invokes a supervisor call that prints the value of `t` on the output console. The output console is an output only device and generates no interrupts. (We ignore the details of print and the console device for the rest of the quiz.)

Name:

The kernel runs each program in its own user-level address space. Each user address space has one thread (with its own stack), which is managed by the kernel:

```
int currentthread;    // index for the current user thread

struct thread {
  int sp;           // user stack pointer
  int pc;           // user program counter
  int pmar;         // user page-map address register
  int r0;           // user r0 register
  int r1;           // user r1 register
} thread[2];  // an array of 2 thread structures

doyield() { // save registers, select new thread, and restore registers
  thread[currentthread].sp = usp;
  thread[currentthread].pc = upc;
  thread[currentthread].pmar = upmar;
  thread[currentthread].r0 = ur0;
  thread[currentthread].r1 = ur1;
  currentthread = (currentthread + 1) % 2;
  usp = thread[currentthread].sp;
  upc = thread[currentthread].pc;
  upmar = thread[currentthread].pmar;
  ur0 = thread[currentthread].r0;
  ur1 = thread[currentthread].r1;
}
```

For simplicity, this thread manager is tailored for the two user threads that are running on Ben's kernel. You may assume in the rest of the quiz that the thread manager must manage no more than two threads.

As you can see, the above implementation of the thread manager is nonpreemptive.

The `yield` call in `main` invokes a supervisor instruction:

```
yield {
  svc 0;
}
```

The `svc` instruction causes the processor to switch into kernel mode and to start executing the following code on the kernel stack:

```
supervisorcall (int n) {
  if (n == 0) doyield();
  rti;
}
```

Name:

Ben's operating system sets up three page maps, one for each user program, and one for the kernel program. Ben has carefully set up the page maps so that the three address spaces don't share any physical memory.

5. [10 points]: How can the current address space be switched?
(Circle ALL that apply)

A. By the kernel writing the `kpmar` and `upmar` registers.

B. By the processor changing the kernel/user mode bit.

C. By the application writing the `kpmar` and `upmar` registers.

D. By `doyield` saving and restoring `upmar`

6. [5 points]: Where can the `rti` instruction in `supervisorcall` resume a user-level thread?
(Circle ALL that apply)

A. in the procedure `supervisorcall`

B. in the procedure `main`

C. in the procedure `yield`

D. in the procedure `doyield`

7. [5 points]: In Ben's design, what mechanisms play a role in enforcing modularity?
(Circle ALL that apply)

A. separate address spaces, because wild writes from one application cannot modify the data of the other application.

B. kernel/user mode bit, because it disallows user programs to write to `upmar` and `kpmar`.

C. the kernel, because it forces threads to give up the processor.

D. the application, because it has few lines of code.

Ben reads about the timer chip in his hardware manual, and decides to modify the kernel to take advantage of it. At initialization time, the kernel starts the timer chip, which will generate an interrupt every 100 milliseconds. (Ben's computer has no other sources for interrupts.) Note that the processor does *not* process interrupts in kernel mode; the processor checks for interrupts only before executing a user-level instruction. Thus, whenever the timer chip generates an interrupt while the processor is in kernel mode, the interrupt will be delayed until the processor returns to user mode. An interrupt in user mode causes an "`svc -1`" instruction to be inserted in the instruction stream, which causes the processor to switch to the kernel and to start executing the following code on the kernel stack:

```
supervisorcall (int n) {
  if (n == -1) dointerrupt();
  if (n == 0) doyield();
  rti;
}

dointerrupt() {
   doyield ();
}
```

You should not make any assumption about how many instructions a processor can execute in 100 milliseconds.

8. [5 points]: Where can the `rti` instruction in `supervisorcall` resume a user-level thread? **(Circle ALL that apply)**

A. in the procedure `dointerrupt`

B. in the procedure `supervisorcall`

C. in the procedure `main`

D. in the procedure `yield`

E. in the procedure `doyield`

9. [5 points]: In Ben's second design, what mechanisms play a role in enforcing modularity? **(Circle ALL that apply)**

A. separate address spaces, because wild writes from one application cannot modify the data of the other application.

B. kernel/user mode bit, because it disallows user programs to write to `upmar` and `kpmar`.

C. the timer chip, because it in conjunction with the kernel forces threads to give up the processor.

D. the application, because it has few lines of code.

Name:

Ben modifies the two user programs to share the variable `t`, by mapping `t` in the virtual address space of both user programs at the same place in physical memory. Now both threads read and write the same `t`. Ben's simple compiler translates the critical region of code:

```
t = t + t;
```

into the processor instructions:

```
100 load t, r0;  // read t into register r0
104 load t, r1;  // read t into register r1
108 add r1, r0;  // add r0 and r1 into r0
112 store r0, t; // store r0 into t
```

The numbers in the leftmost column in this code are the virtual addresses where the instructions are stored in both virtual address spaces. Ben's processor executes the individual instructions atomically.

10. [10 points]: What values *can* the applications print (don't worry about overflows)?
(Circle ALL that apply)

A. some odd number

B. some even number other than a power of two

C. some power of two

D. 1

In an obscure conference proceedings, Ben reads about an idea called "restartable atomic regions", and implements them. If a thread is interrupted in a critical region, the thread manager restarts the thread at the beginning of the critical region when it resumes the thread.

Ben recodes the interrupt handler as follows:

```
dointerrupt () {
   if (upc >= 100 && upc <= 112) {  // in critical region?
      upc = 100;  // restart critical region when resumed!
   }
   doyield ();
}
```

The processor increments the program counter after interpreting an instruction and before processing interrupts.

11. [10 points]: What values *can* the applications print (don't worry about overflows)?
(Circle ALL that apply)

A. some odd number

B. some even number other than a power of two

C. some power of two

D. 1

12. [10 points]: Can a second thread enter the region from virtual addresses 100 through 112 while the first thread is in it (i.e., the first thread's `upc` contains a value in the range 100 through 112)? (Ranges are inclusive.)
(Circle ALL the answers that apply)

A. Yes, because while the first thread is in the region, an interrupt may cause the processor to switch to the second thread and the second thread might enter the region.

B. Yes, because the processor doesn't execute the first three lines of code in `dointerrupt` atomically.

C. Yes, because the processor doesn't execute `doyield` atomically.

D. Yes, because `main` calls `yield`.

Ben is exploring if he can put any code in a restartable atomic region. He creates a restartable atomic region that contains 3 instructions, which swap the content of two variables `a` and `b` using a temporary `x`:

```
100 x = a;
104 a = b;
108 b = x;
```

Ben also modifies `dointerrupt` replacing 112 with 108:

```
dointerrupt () {
   if (upc >= 100 && upc <= 108) {  // in critical region?
      upc = 100;  // restart critical region when resumed!
   }
   doyield ();
}
```

Assume that `a` and `b` start out with the values `a = 1` and `b = 2`, and the timer chip is running.

13. [10 points]: What are the possible outcomes if a thread executes this restartable atomic region and variables a, b, and x are *not* shared?

(Circle ALL that apply)

A. `a = 2` and `b = 1`

B. `a = 1` and `b = 2`

C. `a = 2` and `b = 2`

D. `a = 1` and `b = 1`

End of Quiz I

Name: