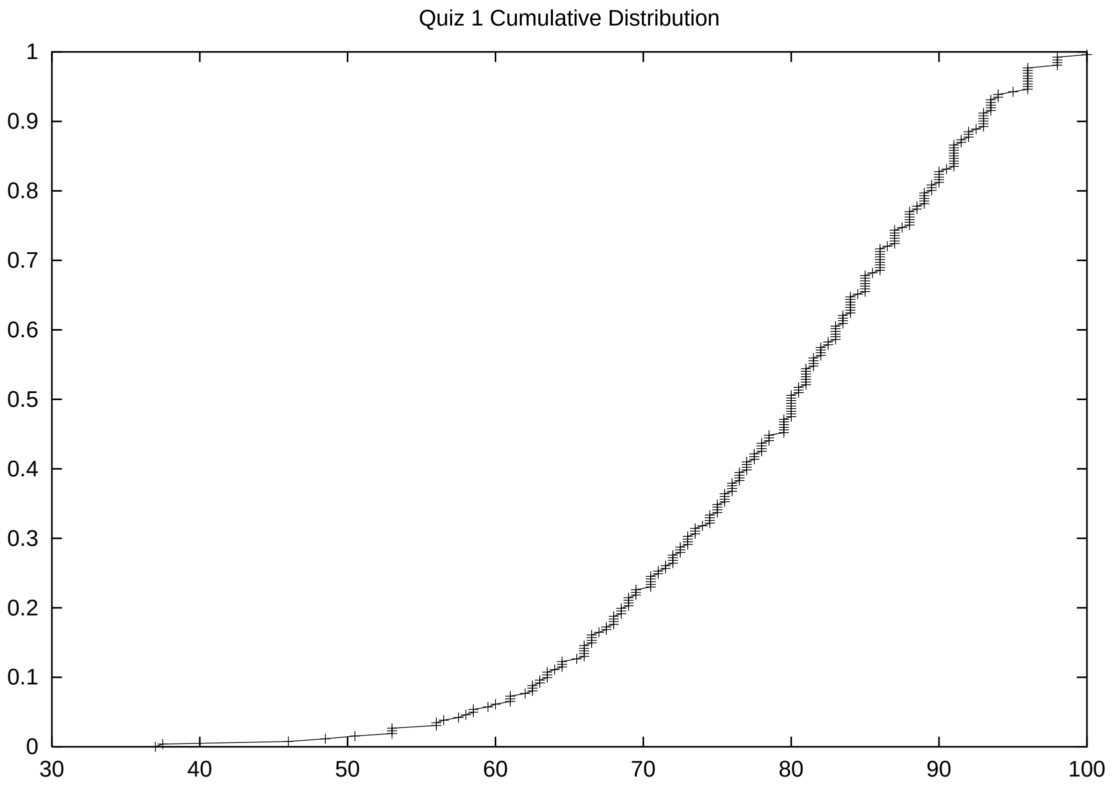
Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.033 Computer Systems Engineering: Spring 2003

Handout 14 - Quiz I Solutions



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.

YES *In many cases, when something failed, there was nothing in the design to stop the failure from turning into a catastrophe.*

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

NO *Hardware interlocks were removed, not added.*

C. Concurrency made anomalous behaviors hard to reproduce.

YES *The system was designed with multiple, concurrent threads and the anomalous behaviors depended on the exact timing of the threads, which was difficult to repeat.*

D. Only enforced modularity was used in the design.

NO *The design was actually not very modular at all. And it used a computer that lacked any way of enforcing modularity on the software.*

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.

NO *PC losering was brought up in a different paper.*

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

NO *UNIX was designed assuming that programs had to fit in a very small address space.*

C. The value of uniform abstractions for devices.

YES *The uniformity of the I/O stream abstraction is a major feature of the design of UNIX.*

D. The data structure design underlying the file system.

YES *Another important feature the paper mentions is that the file system provides a layer of indirection between the file name and the file representation.*

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.

YES AND NO *The paper identified its goal as maximizing the throughput delivered to the network, rather than the throughput of the disk, which implies "no." However the answer might be "yes:" the fact that network capacity exceeds the realized throughput of the disk implies that we should optmize disk performance which is what Flash does.*

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

YES *UNIX provides an asynchronous interface for disk I/O but it is not well supported and provides no way to coordinate disk waits with other waits, so it doesn't help.*

C. All web pages are small.

NO *For one thing, all web pages are not small. For another, Flash is useful whether web pages are small or large.*

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

YES *If the entire website could be cached, there would be no need for Flash.*

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.

See the answer to question 9.9c in the Problems and Solutions section of the 6.033 notes.

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;
}
```

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? *Note that the problem asks about the **current** address space*

(Circle ALL that apply)

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

YES *Writing kpmar changes the current (i.e. kernel) address space. Writing the upmar register does not.*

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

YES *This bit selects whether the kpmar or upmar registers are used.*

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

YES *This was a tricky question: because attempting to store a value in these registers in user mode is illegal, this results in an exception and a switch from user to kernel mode, thus switching the current address space*

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

NO *doyield – in the kernel – saves and restores upmar, which does not switch the current (i.e. kernel) address space.*

6. [5 points]: Where can the `rti` instruction in `supervisorcall` resume a user-level thread? *That is, where does the processing of the user-level thread continue after the* `rti`*?*

(Circle ALL that apply)

A. in the procedure `supervisorcall`

NO `supervisorcall` *is in the kernel and, further, does not call itself.*

B. in the procedure `main`

NO `main` *calls* `yield` *and it is* `yield` *that executes* `svc 0`*. The return from that kernel call will be to the last instructions in* `yield` *not to* `main`*. We're assuming that Ben doesn't have a compiler that inlined* `yield`*.*

C. in the procedure `yield`

YES *The* `svc 0` *returns to* `yield`*, namely to the instruction after* `svc 0` *which is probably a return instruction.*

D. in the procedure `doyield`

NO `doyield` *is in the kernel and does not execute* `svc 0`*.*

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.

YES *This prevents applications from damaging one another.*

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

YES *Without this, programs could modify their address spaces and damage other programs.*

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

NO *The scheduler, as defined so far, is not preemptive.*

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

NO *This has something to do with simplicity, but is not related to modularity.*

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? *That is, where does the user-level thread resume execution after the* `rti`.

(Circle ALL that apply)

A. in the procedure `dointerrupt`

NO *See question 6.*

B. in the procedure `supervisorcall`

NO *See question 6.*

C. in the procedure `main`

YES *Now a clock interrupt can cause an interrupt during the execution of* `main`.

D. in the procedure `yield`

YES *See question 6.*

E. in the procedure `doyield`

NO *See question 6.*

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.

YES *See question 7.*

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

YES *See question 7.*

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

YES *Preemption provides modularity by preventing one thread from dominating the processor and starving other threads.*

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

NO *See question 7.*

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`. *Note that registers are not shared between threads: the scheduler saves and restores the registers on a thread switch.* 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

YES *If* $t = 1$ *and thread 1 gets interrupted after 100, and thread 2 doubles the value of t. Then thread 1 will resume at instruction 104 and end up adding* $t = 1 + 2 = 3$.

B. some even number other than a power of two.

YES *If t=2 and thread 1 gets interrupted after instruction 100, and thread 2 doubles the value of t, then thread one will set* $t = 2 + 4 = 6$.

C. some power of two

YES *This is the behavior if thread 1 is not interrupted.*

D. 1

NO *Neither thread will print the value of t without first doubling it. This assumes that t is statically initialized to 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

NO *The code from 100-112 now executes atomically. So a thread always doubles the value of t, and t starts out as 1 so this produces only powers of 2.*

B. some even number other than a power of two

NO *See answer A.*

C. some power of two

YES *See answer A.*

D. 1

NO *See question 10, part D.*

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.

YES *If the first thread is within the 100-112 region and it gets interrupted, then its pc is set to 100 (which is within the inclusive range 100-112). The second thread can then run and enter the 100-112 region.*

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

NO `dointerrupt` *is executed in kernel mode and in kernel mode interrupts are ignored.* `dointerrupt` *is therefore executed atomically.*

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

NO `doyield` *is also executed atomically by the kernel.*

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

NO *A thread calls yield is only after it is outside the region 100-112.*

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`

YES *If the thread runs without being interrupted the values of a and b are swapped.*

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

NO *No matter how many times the code region is restarted eventually a is assigned the value of b. Therefore a is always set to 2 in the end.*

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

YES *If the code is interrupted between 104 and 108, then a has been set to 2. Now when the code restarts from instruction 100, x gets the new value of a (i.e. 2) and in instruction 108, b gets the value 2.*

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

NO *Same reason as B.*

End of Quiz I

Name: