Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.097 Operating System Engineering: Fall 2002

Quiz I

All problems are open-ended questions. In order to receive credit you must answer the question as precisely as possible. You have 80 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.

1 (xx/35)	2 (xx/20)	3 (xx/25)	4 (xx/15)	5 (xx/5)	Total (xx/100)

Name: Alyssa P. Hacker

I Traps/Interrupts

1. [15 points]: Draw a picture of the kernel stack after the following events have happened: (1) the icode program has called the exec system call; and (2) a clock interrupt handler has interrupted **exec()** at line 3027; and (3) the clock handler just finished executing line 3739? For each entry on the stack describe what that entry represents. You may ignore local variables.

(Draw high addresses high.)

The kernel stack starts off empty. A system call must be the first thing on a trap frame at this point — if any interrupts happened before, they would not have had any other process to switch to on the return path.

in trap	PSW from icode
	PC of icode's SYS instruction
	r0 of icode
	PSW after trap (br7 + 6)
	r1 of icode
	SP of icode
	masked PSW after trap (6)
	return address for call1 return (line 786)
in _trap	saved r5
	saved r4
	saved r3
	saved r2
	&cret
	&exec (an argument to trap1)
	return address in _trap (line 2772)
in _trap1	saved r5/r4/r3/r2 + &cret
	return address in _trap1 (line 2848)
in _exec	saved r5/r4/r3/r2 + &cret
CLOCK interrupt	PSW from exec
	PC from exec (line 3027)
	r0 from exec
	PSW after interrupt (br6)
	r1 from exec
	sp (same as sp now)
	masked PSW
	return address in trap (line 800)
in _clock	saved r5/r4/r3/r2 + &cret

2. [5 points]: Which kernel stack is the one you drew in response to question 1? Kernel stack 0? Kernel stack 1?

(Explain your answer briefly)

Kernel Stack 1 is used. Process 1 (icode) was executing at the time of the trap and so its kernel stack is used. If the machine is already in kernel mode when an interrupt happens, the current kernel stack is used.

3. [5 points]: Can another clock interrupt interrupt the handler at line 3739? At line 3770?

(Explain your answer briefly)

At line 3739, another clock interrupt *can not* interrupt the handler: the priority of the processor has been set to 6 already as a result of line 534 (or 535).

At line 3770, the interrupt handler has lowered the priority level to 5 so another clock interrupt (at level 6) *could* interrupt the handler at this point.

4. [10 points]: Estimate how large a kernel stack must be to handle recursive traps and interrupts correctly?

(Explain your answer briefly)

The kernel stack must be able to hold sufficient stack and trap frames for the most recursive case. This occurs if a process executes a system call, receives a priority 4 interrupt, followed by a priority 5 interrupt, followed by a priority 6 interrupt, followed by a re-entrant priority 6 interrupt.

In addition to the size of the trap frame (i.e. the arguments to **_trap**), you would need to consider stack frames generated by the deepest system call procedure call path (e.g. **open** calls **namei** which calls . . . ; this is probably not too deep but might change as the kernel evolved), the deepest interrupt 4 handler call path, etc.

From the figure, we can see that a system call trap requires 20 stack elements for the trap frame and stack frames for the trap dispatch code (plus some extra for locals). Similarly, a non-syscall trap requires 8 stack elements for the trap frame. Assume that the maximum call depth for a function in the kernel is 4, where each stack frame requires 6 stack elements (5 from **csav**, and an average of 1 argument). This gives a maximum depth of approximately

$$20 + 4 \times 6 + 4 \times (8 + 4 \times 6) = 172$$

. Since each stack entry is two bytes, this gives a lower bound of 344 bytes. Of course, some additional room will be needed to account for local variables, which were ignored for this problem.

II Coordinating threads

5. [5 points]: What do the statements at line 2088 and line 2092 do?

(Explain your answer briefly)

Line 2088 calls **spl6()** which sets the processor privilege level to 6. This causes all interrupts of priority less than or equal to 6 to be ignored. Line 2092 calls **spl0()**, lowering the privilege level back to 0. These calls are used to mark atomic sections.

6. [15 points]: Why are these statements needed? Give an actual sequence of events that will result in erroneous behavior if these two statements are omitted. Your sequence of events should be as concrete as possible, not merely hypothetical.

(Illustrate your answer using a time line)

Several scenarios can lead to incorrect behavior:

Deadlock:

1. **runin++** on 1954
2. scheduler calls **sleep** on 1955 with **runin** as channel
3. executes lines 2089 setting **p_wchan** to **&runin**
4. processor is interrupted by clock
5. clock interrupt handler sets **runin** to zero (3821)
6. handler calls wakeup with **&runin** as argument (3822)
7. wakeup calls setrun, which sets **p_wchan** to zero and **p_stat** to **SRUN** (2139–2140)
8. handler returns
9. now scheduler sets **p_stat** to **SSLEEP** (2090)
10. and calls **swtch**
11. scheduler will never be woken up, because **p_stat == SSLEEP** and **p_wchan** is zero

Deadlock:

1. a user process calls indirectly **bwrite()**, which launches write on buffer.
2. then calls **iowait**,
3. which calls sleep with the address of the buffer as the channel
4. sleep sets **p_wchan** (2089)
5. disk interrupt comes in completing the write of the buffer
6. it calls **iodone**, which calls **wakeup** on the buffer address
7. calls **setrun**, which sets **p_wchan** to zero and **p_stat** to **SRUN**
8. interrupt handler completes
9. process sets **p_stat** to **SSLEEP** (2090)
10. and call **swtch**
11. process will never be woken up because **p_wchan** is zero and **p_stat SSLEEP**

Name: Alyssa P. Hacker

III Address spaces

7. [10 points]: Draw the prototype segmentation registers with their actual content after the call to **estabur()** on line 3152 has completed. Assume that **w0** is 410, **w1** is 129, **w2** is 32, and **w3** is 128 (see lines 3076 through 3082). Explain the content of the registers briefly.

(Ignore the protection bits in the descriptor)

Converting what we are given into blocks (and hence the arguments to **estabur**), we get:

```
text size is 129 bytes (w1 = 129)          ==> nt = 3 blocks
SSIZE (stack size) = 20 blocks             ==> ns = 20 blocks
data size is 160 bytes (w2 = 32, w3 = 128) ==> nd = 3 blocks
USIZE = 16 blocks
```

That is, **estabur(nt = 3, nd = 3, ns = 20, sep = 0)**.

Since each segment (8KB) contains 128 blocks, the text, data, and stack each fit within one segment:

```
          +------------+          +------------+
PAR[7]:   |    -89     |  PDR[1]: | 108 ED RW  |
          +------------+          +------------+

PAR[7] = USIZE + nd + ns - 128
       = 16 + 3 + 20 - 128
       = 39 - 128
       = -89

PDR[1] = (128 - ns) << 8 | ED | RW
       = 108<<8 | ED | RW

          +------------+          +------------+
PAR[1]:   |  USIZE     |  PDR[1]: |  2      RW |
          +------------+          +------------+

PAR[1] = USIZE = 20
PDR[1] = (nd - 1) << 8) | RW
       = 2<<8 | RW

          +------------+          +------------+
PAR[0]:   |     0      |  PDR[0]: |  2      RO |
          +------------+          +------------+

PAR[0] = 0, because at line 1675 a = 0
PDR[0] = (nt - 1) << 8) | RO
       = 2<<8 | RO
```

All other PARs and PDRs are cleared by lines 1694–1696.

8. [10 points]: Why is "a-128" stored in the prototype segment for the stack? Explain your answer by showing how the virtual address 0177777 is translated to a physical address.

(Explain your answer briefly)

At the point where "a-128" is written into PAR[7], "a" equals the combined sizes of the u area, data, bss and stack. A contiguous region of physical memory ("a" blocks in size) is allocated to hold these components of the process.

```
+-----------+
|   stack   |
+-----------+
| data+bss  |
+-----------+
|  u area   |
+-----------+

a blocks in size
```

Dividing the VA 0177777 into its constituent fields, APF:block_number:block_offset yields 7:127:63. Translation of this VA selects PAR[7] which holds "a-128". Added to this number is the block_number, 127, resulting in physical block number "a-1". The physical address refernced is therefore "a-1":63. But since the chunk of physical memory allocated was of size "a" blocks, this precisely refers to the last byte (byte 63) of the last block (block "a-1"). This is where the first byte of the stack is located.

9. [5 points]: What happens if the processor executes the instruction "tst -(sp)", and the user stack pointer falls below the stack segment? Write down the names of the functions that v6 invokes in order.

(Explain briefly)

A segmentation exception will result and the stack will grow (assuming free memory can be found).

The following calls will be made:

```
Machine generates exceptions
  => trap:
     => trap()
        => backup()
        => grow()
```

Then control returns back to the user process and it resumes execution where it left off, re-executing the "tst -(sp)" instruction. The second execution succeeds of course.

IV I/O and file systems

10. [5 points]: Why does the structure **struct file** exist? Why not have a file descriptor point directly to the incore inode structure, and store **f_offset** in it?

struct file exists so that if two separate processes open the same file, they have their own offset pointer into it.

11. [10 points]: Assume the directory "d" exists, that it is the current working directory, and that it has two entries (including "." and '"..")。 If an application calls **creat ("bar", 0444)**, and the power fails right after the processor completed the call **wdir** on line 7467, what is the state of the file system on disk? More specifically, does the file "bar" exist on disk? Does the file "bar" show up in the directory "d" on disk?

(Be brief)

The updated inode of directory "d" exists (because inodes are updated synchronously). The data of "d" may not be written because v6 wrote it using **bdwrite** (it is a partial block). The file doesn't exist yet, because we haven't gotten far enough yet into the **creat** call (**maknode** is called from line 5790).

End of Quiz I Solutions