Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.828 Operating System Engineering: Fall 2006

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/20)	2 (xx/10)	3 (xx/25)	4 (xx/35)	5 (xx/10)	Total (xx/100)

Name: Solution Set

Quiz score statistics: median 65, mean 67, maximum 100.

I Processes and concurrency

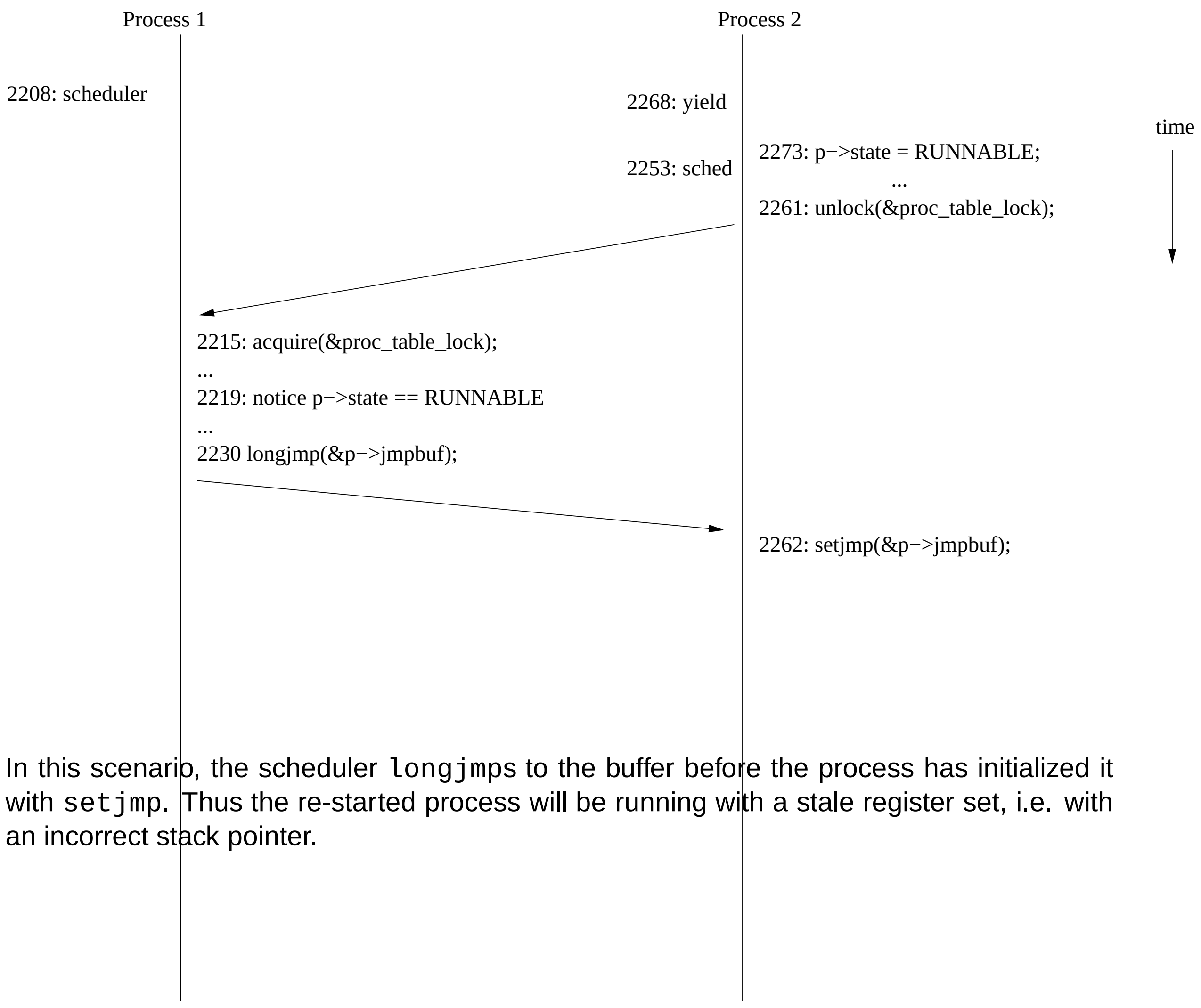
1. [10 points]: Ben Bitdiddle observes that releases and acquires of `proc_table_lock` are spread through proc.c, which makes it difficult to reason if the code is correct. He rereads the comment on line 2222, and proposes that the to-be-scheduled process doesn't release `proc_table_lock` when it runs. That is, he removes all the `acquire` and `release` calls for `proc_table_lock` except the two already in `scheduler`. He recompiles xv6 and boots it. What will Ben observe?

Only one processor at a time will be able to run a user process.

2. [10 points]: Dissatisfied with the simplification of the previous question, Ben Bitdiddle explores another altnerative. He undoes all his changes, and now he modifies the code so that during a context switch from one process to another, `proc_table_lock` isn't held. That is, he makes the following changes:

- release `proc_table_lock` before line 2262 and before line 2229;
- acquire at 2231;
- remove the releases in, for example, `yield` and `sleep`.

Ben observes a race condition between one processor running `scheduler` and a second processor running `yield` (on behalf of a user process). Give a timeline of events with 2 processes that demonstrates the race condition.



Process 1
Process 2
2208: scheduler
2268: yield
time
2273: p−>state = RUNNABLE;
...
2253: sched
2261: unlock(&proc_table_lock);
2215: acquire(&proc_table_lock);
...
2219: notice p−>state == RUNNABLE
...
2230 longjmp(&p−>jmpbuf);
2262: setjmp(&p−>jmpbuf);

In this scenario, the scheduler `longjmp`s to the buffer before the process has initialized it with `setjmp`. Thus the re-started process will be running with a stale register set, i.e. with an incorrect stack pointer.

Name:

II System calls

3. [10 points]: If one process kills a process that is sleeping, it calls `proc_kill`, which sets `p->killed` and makes the target process runnable. In `trap`, the target process checks `p->killed` and exits. An alternative is for the target process to check `p->killed` when it comes out of sleep (e.g., right after line 2230), and call exit. Why is this alternative undesirable? That is, why wait with exiting until the return from trap? (Hint: think about system calls that invoke sleep several times, such as creating a file). Be specific in your answer by giving a concrete example.

If the process has any data structures in inconsistent states, it needs to be given a chance to undo them.

For example, at line 4024, if the process stopped executing during `ide_rw`, the block it is holding would never be `brelse`'d (would be `B_BUSY` forever). Other processes trying to read this block later would deadlock.

III Address spaces

4. [10 points]: JOS uses the paging support of the x86 processor, while xv6 uses only the segmentation support. Clearly using paging in JOS required you to write more lines of code than xv6 requires for using segmentation. List 3 significant ways in which JOS is better than xv6 because of its use of paging.

JOS does not require an environment's memory to be laid out in contiguous physical memory.

JOS can allow sharing of memory between environments at small granularity (individual pages). This is used, for example, to share the UENVS table with all environments.

JOS can edit memory permissions at small granularity (individual pages). This is used, for example, to implement copy-on-write fork.

(There are other correct answers as well. Some of these benefits could be realized with segmentation, but not the way xv6 uses segments, and probably not using C.)

Name:

In JOS `page_free` is called when the reference count of a page reaches 0. It adds the page to the free list of pages. Of course, the kernel can reference the page via the mapping at `KERNBASE` (the one used by `KADDR`).

5. [10 points]: To keep the kernel honest you might consider removing the `KERNBASE` mapping in `page_free`. Assuming no other changes outside of `page_free`, what would break as a result?

After a page is freed, accesses to the `KADDR` mapping would cause extra page faults once the page was re-allocated and referenced by the kernel (for example, to zero the page before reusing it).

6. [5 points]: How would you fix this problem while still removing the `KERNBASE` mapping in `page_free`?

Remap the page in `page_alloc`.

Name:

IV File systems

The xv6 cache implements a least recently-used cache replacement policy (i.e., it evicts the block from the cache that hasn't been used for the longest time).

7. [4 points]: What lines of code reorder the `bufhead` list to reflect the buffers' usage pattern?

4037–4042

8. [6 points]: List the code changes necessary to make xv6 implement a most-recently used policy (i.e., evict the block from the cache that has been used most recently). (Give line numbers that would need to change, along with the changes.)

Change line 3987 to traverse the list in the opposite direction.

(Or change lines 4037-4042 to insert the released block at the other end of the list.)

9. [10 points]: Identify one disk block that you expect to be in the cache frequently with the least-recently used policy. (Explain briefly.)

The superblock (block 1) is used in every call to `balloc`, `bfree`, and `ialloc`.

(There are other correct answers as well.)

In xv6 two child processes can communicate through pipes only if they have a common ancestor who created the pipe. To support communication between any two processes, some Unix operating systems provide *named pipes*, which are also called FIFO. A named pipe is durable and appears in the file system name space, and must be unlinked explicitly after use. One could use named pipes as follows. One process could do as follows:

```
mkfifo pipe       // make a named pipe
gzip < pipe > out // read from the named pipe, compress, and write to out
```

And, another process, unrelated to the first, could do as follows:

```
cat file > pipe   // write the contents to the named pipe
```

The result of opening a named pipe is a file descriptor that behaves exactly like one of the two file descriptors returned by the pipe call, depending on the open mode (for reading or for writing).

10. [15 points]: Write up how you would change xv6 to support named pipes. List the functions that would have to be added or changed, with short descriptions of the new functionality or changes.

Add a new system call `sys_mkfifo` that uses `mknod` to create an inode with the new type `T_PIPE`.

Create a new in-memory table mapping inode numbers to `struct file*` pairs.

In sys_open, at line 5285, check whether `ip->type` is `T_PIPE`. If so, look in the table for an entry for `ip->inum`. If there is not one there, allocate a new table entry and call `pipe_alloc` to get a pair of `struct file*` for the new entry. Use the corresponding `struct file*` (either the read or write end) instead of calling `filealloc` at line 5286.

Name:

V 6.828

11. [5 points]: Describe the most memorable error you have made so far in one of the labs. (Provide enough detail that we can understand your answer.)

Some memorable bugs (in summary):

Using `M-x comment-region` to comment out assembly in Emacs commented the lines with `;`, which does not actually introduce a comment in x86 assembly.

Setting the upcall function pointer to the C page fault handler instead of the assembly stub.

Pointer arithmetic problems, especially adding a `PDX` to an integer address instead of to a `pte_t*`.

Adding ANSI text coloring but not changing the default (code zero) black on black.

We would like to hear your opinions about 6.828, so please answer the following two questions. (Any answer, except no answer, will receive full credit!)

12. [3 points]: What is the best aspect of 6.828?

13. [2 points]: What is the worst aspect of 6.828?

End of Quiz I

Name: