CS140
Operating Systems and Systems Programming

Final Exam. Summer 2006.
By Adam L Beberg.
Given August 19th, 2006.

Total time = 3 hours, Total Points = 335

Name: (please print)____

In recognition of and in the spirit of the Stanford University Honor Code, I certify that I will neither give nor receive unpermitted aid on this exam.

Signature:____

- This exam is closed notes and closed book.
- No collaboration of any kind is permitted.
- You have 3 hours to complete the exam.
- There are 24 questions totaling 335 points. Some questions have multiple parts. Read all parts before answering!
- Please check that you have all 13 pages.
- Before starting, write your initials on each page, in case they become separated during grading.
- Please print or write legibly.
- Answers may not require all the space provided. **Complete but concise answers are encouraged.**
- SCPD students: If you wish to have the exam returned to you at your company, please check the box.

Page 2	/10
Page 3	/30
Page 4	/30
Page 5	/25
Page 6	/25
Page 7	/35
Page 8	/40
Page 9	/35
Page 10	/40
Page 11	/30
Page 12	/35
TOTAL	**/335**

SCPD?	

INSTRUCTIONS

Near the beginning of the course, if not the first lecture, I said that even the "Hello World" application involved millions of lines of code to make run. Now that the course is over, and with your experience with Pintos, you should understand what all of those lines of code are doing.

This is your chance to demonstrate that you understand operating systems. If you watched the lectures, did the assignment, and read the materials, all of these questions should be easy. This exam is 1/3 of your grade, so show us you know the material. You should have much more time then you need to do a great job.

All questions will refer to the source code on the LAST page of this exam as "the code". Feel free to rip that page off and refer to it, and use it as scratch paper, you do not need to turn that page in (we have a copy).

READ ALL THE QUESTIONS FIRST! Unlike most tests, this is not just a suggestion. In general you're asked what happens when things go smoothly, then about when things go a different way, each questions follows from the one before. If you're not about to be asked about an edge case, or an important issue, go ahead and mention it however.

BEGIN CS 140 FINAL EXAM

1. [10] You take the code and run `` `gcc -c final.c -o final.o` `` to compile the code. What stuff is in the final.o file? Exact names/number are not needed, you did that in the midterm.

   Main things you should mention:

   The compiled code (text segment), but with none of the addresses filled in.
   The data segment, with initialized data.

   Table of data definitions for all declared variables - defs.
   Table of code references to external and internal functions- refs.

2. [10] (a) Next you link final.o along with the standard C library. What does the linker spit out if the C library is a static library - libc.a?

   The linker puts together all the text (code) segments into one big text segment. Same for data, etc.

   Takes all the defs and refs and resolves them, inserts real addresses into all those locations.

   Puts a nice header on the resulting segments, with the start address and lots of descriptive info, and spits it out.

   [5] (b) What unused code will end up in the program, and why?

   Extra unused functions end up in the end binary because the linker isn't smart enough to only link in the functions it needs, but also puts any functions in the same .c files as functions it needs in.

3. [10] Same as question #2, but with a dynamic stared library - libc.so, how is this program different?

   Instead of adding in the code, and resolving all the addresses, it instead puts in jumps/pointers to addresses in a table that are filled in at run time. Glue logic is also added to load the library and it's data and fill in the table.

   If linking is delayed all the way to call time, then the table is filled in the first time that function is called, not when the program is loaded.

4. [5] List 3 things in the ELF header(s) of the program.

   Many things, too many to list here. Search for "Executable and Linkable Format" or do a `` `man elf` ``

   Common answers were the start address, segment start/sizes, CPU/OS types, etc.

5. [20] (a) Now you run `` `final` ``. What does the operating system do up to the point where you add it to the ready list? Assume lazy loading like in the project. (completely ignore the details of the file system for now, we'll get to that)

   Some of the main points:

   Load the header, and get the various segments and offsets from the file.

   Allocate a process - PCB, (and/or thread if they are the same) for the priority, owner, etc.

   Add page table entries for the segments of the file we need, and marked them as not-present.

   Allocate and setup a stack, parse arguments, make it look like it's coming back from an interrupt, and returning to the start address.

   Put the thread on the ready list (implied by question).

   [5] (b) And if we weren't so lazy?

   The entire program would be loaded into physical memory before we add the process to the ready list. This would probably involve loading in more pages then we may (ever) need to use, and thus evict more then we have to from other processes.

6. [5] When you run final, all physical memory is already in use. What happens?

   The OS will have to evict another process's pages before it can load ones for the new process. It will do this with some algorithm that best or most easily approximates the impossible MIN(), so an LRU like the clock algorithm etc.

   The key point was to mention something about how you picked the victim page.

7. [5] List 2 situations where the OS will not let the program run for you at that moment, but would let you if the situation was not happening.

   Some possible answers:

   OS/User process limit reached.
   OS/User memory limit reached.
   Machine is thrashing and not letting in any new process.

8. [5] (a) Finally everything is going according to plan, you're ready. When do you get to run?

   The scheduler has to pick your thread to run, taking you from the READY list to the RUNNING state (only one per CPU, not a list).

   [5] (b) Why could this be a while?

   Lots of threads and along timeslice in RR.
   Big job "convoy" effect in FCFS.
   Low priority in multi-queue algorithms with no aging of low priority processes.
   Too long in a STCF system with lots of new short jobs all the time.

   [10] (c) If this was a typical type of program on the machines you administered, what scheduling algorithm would you want to use? Justify your choice.

   Due to the high I/O, then CPU, then more I/O we want an algorithm that is adaptive to changing workloads, so that we can keep CPU and I/O both busy. Multilevel feedback queue is probably the best choice for that reason.

   Some relation to the code was needed in your answer, not just picking multilevel feedback queue.

9. [5] (a) Which lines of code *always* generate a systems call? List #'s.

   5, 7, 9 - file operations

   11, 12, 13, 14 - networking operations

   [5] (b) Which lines of code *might* generate a system call? List #'s.

   6 - malloc() may need a new page.

   10 - free() may return pages.

10. [10] (a) Assume timeslices end right after every line of code by using the Pyschic Timeslice Algorithm™. If we're using an exponential feedback algorithm, what's happening to our priority as we go through the program. Use lower to mean low priority in your description, not a low queue number, and higher to mean higher priority.

1	1-4 are not run	8	lower
2	...	9	higher
3	...	10	higher
4	...	11	higher
5	higher	12	higher
6	higher or lower OK	13	higher
7	higher	14	higher

   I/O and fast things makes it higher, long computation makes it lower.

   [5] (b) What if we're billing all time spent in the system call time to the process, would this change your answers? Which ones?

   5 - open() can get ugly with enough permissions etc.

   6,10 - malloc/free can be long if much of memory is full, bookkeeping.

   Arguments can be made for 7 & 9 taking a long time with enough fragmentation.

   11-14 are still just commanding devices and moving on.

   You should have had at least 1 or 2 of these reasons.

11. [20] (a) Now back to that file system were ignoring. If we're running on the Elaine machines and using an ACL based access control system, what happens when the open() in line 5 is run? Describe all the structures and OS ideas involved. (now you can ignore the syscall details)

First, the filename is parsed out into each directory and the filename.

Starting at the root directory, which is at a known location, the system loads up the inode of the directory, and then the directory and it's ACL, then checks them against the user.

It then checks for the next directory, "foo" in this case, and repeats.

When it gets to the file part of the path, we look for it in the last directory, and if so, we locate it's inode.

Once all those permissions have passed, we can store this inode, give it a per-process number, and use that number to service further syscalls on the file.

[10] (b) What if we were in a Capabilities based system? Also, What happened at login to make this possible?

Instead of the permissions checking against the list of allowed users on the directory/file, the system checks the object against the users list.

At login, your capabilities were assigned to you, and your login process.

[5] (c) What if the file isn't openable, what's different between how ACL and Capabilities handle it?

When you don't have access to something in a capabilities system, it's completely hidden from you.

12. [10] Now we're in a typical UNIX file system with inodes. What's going to be happening as far as figuring out where the blocks are as we read (many) blocks due to line 7.

    The inodes must be looked at the get the physical block numbers that the files data resides in.

    This will first involve the direct blocks, which are part of the inode.

    For longer files, this then leads to indirect blocks, which means you have to read another disk block full of pointers to blocks.

    Double-indirect blocks come into play after one block's worth of pointers, and then you traverse 2 layers to get to the block pointers.

    [5] (b) … and with an extent based file system?

    With extent, you have a series of (start, length) pairs, and blocks are laid out sequentially. This makes the messy inode structure/traversal unnecessary.

13. [10] How would this be different if before line 7 there was a call to mmap().

    Instead of blocks going into the buffer cache, the VM system is told to map the file's blocks into virtual memory, and from there on, the VM system pages in/out blocks as if it were a special swap file.

    This is generally much more efficient.

14. [10] (a) By now we know what disk blocks we need and have a long list, and so do other processes (maybe they have short lists). What do we tell the disk to do so everyone is happy?

    You need to schedule the disk accesses in a way that is fair and fast. We talked about the elevator algorithm that scans from the first to the last cylinder and reads/writes blocks as it goes. This way noone starves, but the disk is also busy.

    [5] (b) What does SCSI do that IDE doesn't that's makes this a non-issue.

    SCSI reorders the read/write requests for you, so you don't have to.

15. [10] After we start this program, the system comes to a standstill, and becomes unbearably slow. In what ways could `BIG_NUMBER` be involved in this.

    Large amounts of memory are used, so we could trigger thrashing.

    Large amounts of disk I/O may grow the buffer cache and force memory to swap.

    [5] (b) Is this really avoidable at all if `BIG_NUMBER` is really big? How? Your answer should not involve violence ;)

    There were many possible answers for this, but the short answer is no, but you can make it less bad.

    You can try by buying memory, impose memory limits, kill processes, schedule using working sets, etc.

16. [10] (a) Is a log based file system likely to help this processes performance or hurt it or not matter? Why?

    Since this program does just a large sequential write, and reads are essentially the same in a log file system, the overhead of the double-writes hurts you.

    If the log was on another disk you may not notice much.
    If you were just logging the metadata you also would not be much slower.

    It means you won't crash as bad of course.

    [10] (b) What would happen differently in line 7 and in line 9 in the case of a log based file system?

    Line 7 is pretty much unaffected. Logging cares only about sequences of writes we're not done with yet.

    Line 9, the writing is changed greatly. Everything we write first goes to the log, along with information on what we are doing, then the real file system. This allows us to do caching, and only do the real-write at the end.

17. [5] How do we battle latency and bandwidth limits in disk systems?

    Latency is helped by read-ahead and caching.

    Bandwidth can be helped by doing good disk scheduling, and with RAID.

18. [10] (a) Alice is sending Bob a message in a public key cryptosystem. What does she do in which order.

    Alice signs the data with her private key which only she has. Then encrypts the data with Bob's public key which only Bob can undo. The other order leaves Alice's identity open, and Homeland Security will come to visit.

    [5] (b) What does Bob do when he gets the message?

    Decrypts the message with his private key, so only he (and the rootkit on his machine) can read it. Then he checks the signature from Alice, which only she could have made... unless Homeland Security has already been there and used the 4 B's.

    [5] (c) If Alice and Bob have never talked directly to each other in person or by phone, is this system secure?

    No. Alice doesn't know if Bob's public key is really his. And Bob doesn't really know about Alice's key either.

19. [10] (a) What happens during the connect() call in line 12?

    The 3 way handshake.

    1. We send an (open, seq #x)

    2. The other side sends us an (ACK x, seq #y)

    3. We reply with an (ACK y)

    [5] (b) If it was a UDP socket instead?

    Nothing happens except the default address get set.

20. [20] (a) Draw a diagram of what the packet sent in line 13 will look like going across the wire. You don't need to remember all the exact header fields, just the ones we highlighted in class. You _DO_ need every extra chunk added between line 13 and the wire. Label each addition as belonging to the end-to-end, network, or link layer. Here is a start: (since the page is only 8.5 inches wide, some arrows may be useful)

Looking for something along these lines, as long as you got the 3-layer wrapping right, and the main fields, it's probably OK.

Link Layer: Ethernet Header
Sentinel, source ethernet address, dest addr

Network: IP Header
Source IP, dest IP, length, header checksum

E-to-E Layer: TCP Header
Source port, dest port, offset, seq, ack, header+data checksum

data[128 bytes] (from user)

Link Layer: checksum, sentinel

21. [10] What 4 packet problems does TCP fix, and what does it use to solve them all.

Lost Packets, Lost ACKs, reordering of packets, reappearing old packets.

All 4 are solved with ACKs, retry timers and the sequence number, which is a combination of a unique stream ID and the packet number.

Also accepted: does sliding window for flow control.

22. [5] (a) Why do wireless networks normally use encryption in the link-layer, in clear violation of the end to end principle.

Because without wires, anyone can be listening, jamming, or using the connection.

Variations on that theme.

[5] (b) Why _doesn't_ encryption happen in the link later, since we should all obviously be using it for everything now that we're all being watched at all times.

Because encryption is done at the application layer when it is needed, and so this would be redundant.

23. [10] (a) Buffer overflows… what stupid thing do programmers keep doing even though it's 100% obvious when you look at the code.

Putting buffers on the stack. Fixed by using malloc, so the buffer is on the heap.

Also acceptable but not in the code: not checking bounds on all array/string operations.

[5] (b) What did the hardware folks do (or stopped doing for some reason) that made this a problem in the first place?

Data on the stack is mixed in with return addresses.

The stack is executable.

24. [10] Pick 2 lines of code that happen differently (but act the same of course) if we're in a Virtual Machine Monitor. Briefly, what's different?

Any line that did a system call was fair game. The summary is that this will trap into the VMM, and then have to be simulated as an effect on the guest OS, while the VMM actually controls the real hardware/memory/device.

END CS 140 FINAL EXAM

“THE CODE”

Assume during the "`...`" that any undeclared variables are declared, any variable that need initializing are initialized, and all function calls succeed. Those details are not important.

```
Line #   | final.c
         |
         | int main( int argc, char * argv[] )
         | {
1:       |   int file;
2:       |   int socket;
3:       |   char packet[128];
4:       |   void * matrix = NULL;
         |   ...
5:       |   file = open( "/tmp/foo.txt", "r+" );
         |   ...
6:       |   matrix = malloc( BIG_NUMBER );
7:       |   read( file, matrix, BIG_NUMBER );
8:       |   do_matrix_invert( matrix );
9:       |   write( file, matrix, BIG_NUMBER );
10:      |   free( matrix );
         |   ...
11:      |   socket = socket( PF_INET, SOCK_STREAM, 0 ); /*TCP*/
12:      |   connect( socket, &address, sizeof(address) );
13:      |   send( socket, &packet, sizeof(packet), 0 );
14:      |   len = recv( socket, &packet, sizeof(packet), 0 );
         |   ...
         | }
```