The University of Texas at Austin
CS 372H Introduction to Operating Systems: Honors: Spring 2010

FINAL EXAM

- This exam is **3 hours**. Stop writing when "time" is called. *You must turn in your exam; we will not collect them.* Do not get up or pack up in the final ten minutes. The instructor will leave the room 3 hours and 3 minutes after the exam begins and will not accept exams outside the room.
- There are **28** questions in this booklet. Many can be answered quickly. Some may be harder than others, and some earn more points than others. You may want to skim all questions before starting.
- **This exam is closed book and notes. You may not use electronics: phones, PDAs, calculators, etc.** You may refer to TWO two-sided 8.5x11" sheet with 10 point or larger Times New Roman font, 1 inch or larger margins, and a maximum of 60 lines per side.
- If you find a question unclear or ambiguous, be sure to write any assumptions you make.
- Follow the instructions: if they ask you to justify something, explain your reasoning and any important assumptions. **Write brief, precise answers. Rambling brain dumps will not work and will waste time.** Think before you start writing so you can answer crisply. Be neat. If we can't understand your answer, we can't give you credit!
- To discourage guessing and brain dumps, we will, except where noted, give 25%-33% of the credit for any problem left *completely blank*. If you attempt a problem, you start at zero points for the problem. Note that by *problem* we mean numbered questions for which a point total is listed. *Sub-problems* with no points listed are not eligible for this treatment. Thus, if you attempt any sub-problem, you may as well attempt the other sub-problems in the problem.
- The exception is the True/False problems. There, we grade by individual True/False item: correct items earn positive points, blank items earn 0 points, and incorrect items earn negative points. However, the minimum score on any question—that is, any group of True/False items—is 0. Don't overthink all of this. What you should do is: if you *think* you *might* know the answer, then answer the problem. If you *know* you *don't* know the answer, then leave it blank.
- Don't linger. If you know the answer, give it, and move on.
- **Write your name and UT EID on this cover sheet and on the bottom of every page of the exam.**

Do not write in the boxes below.

I (xx/12)	II (xx/13)	III (xx/15)	IV (xx/19)	V (xx/16)	VI (xx/17)	VII (xx/8)	Total (xx/100)

Name: Solutions **UT EID:**

I Interrupts, concurrency, scheduling (12 points total)

1. [2 points] What are the three ways that a CPU executing in user mode (that is, executing instructions from a user-level process) can switch to supervisor mode (and presumably execute kernel instructions)? (Hint: we have discussed this question in class, using the phrase, "What are the three ways that the kernel gets invoked?")

A. Interrupts from a device.

B. Traps, often used for invoking system calls. Confusingly, on the x86, the `int` instruction generates traps.

C. Exceptions, such as divide by zero, illegal memory access, or execution of an illegal opcode.

2. [2 points] Pat Q. Hacker from the midterm is back. Pat gets a job writing an operating system. Pat is told by the company's marketing department that the kernel needs to be correct only on a machine with a single CPU. Pat's manager imposes a further requirement: it is unacceptable to write a kernel in which hardware interrupts are always disabled (this manager really dislikes the approach that JOS takes). Given these requirements, Pat decides to leave interrupts on *all* the time. Pat's kernel has data shared between the device interrupt handlers (meaning the code that runs when an external I/O event happens) and other kernel code. Pat reasons that no special steps to protect this shared data are needed because a single CPU can do only one thing at a time.

State whether Pat's reasoning is correct, and justify your answer *briefly* below:

Pat's reasoning is incorrect. If Pat's kernel code is interrupted, then the interrupt handler might run and invalidate an invariant (for example, imagine that the other kernel code is fixing up a linked list, and the interrupt handler needs to enqueue to this linked list). To get around this, Pat needs to protect data that is shared between the interrupt handlers and other kernel code. To do so, Pat probably needs to disable interrupts when messing with data that the interrupt handler also modifies. Doing so ensures the invariant that if the interrupt handler is modifying the data, then the other kernel code is not (and vice-versa).

3. [8 points] In this problem, you will implement a monitor to help a set of *drivers* (modeled as threads) synchronize access to a set of *five keys* and a set of *ten cars*. Here is the problem setup:

- The relationship between the keys and the cars is that key 0 operates cars 0 and 1, key 1 operates cars 2 and 3, etc. That is, key i works for cars $2i$ and $2i+1$.
- If a key is being used to operate one car, it cannot be used to operate the other.
- A driver requests a particular car (which implies that the driver needs a particular key). However, there may be many more drivers than cars. If a driver wants to go driving but cannot get its desired car or that car's key, it waits until the car and key become available. When a driver finishes driving, it returns its key and notifies any drivers waiting for that key that it is now free.

- You must allow multiple drivers to be out driving at once, and you must not have busy waiting or spin loops.
- We repeat: there could be many, many instances of `driver()` running, each of which you can assume is in its own thread, and all of which use the same monitor, `mon`.

On the next page, fill in the monitor's remaining variable(s) and implement the monitor's `take_key()` and `return_key()` methods. ***Follow the coding standards given in class.***

```
typedef enum {FREE, IN_USE} key_status;

class Monitor {
    public:
        Monitor() { memset(&keys, FREE, sizeof(key_status)*5); }
        ~Monitor() {}
        void take_key(int desired_car);
        void return_key(int desired_car);
    private:
        Mutex mutex;
        key_status keys[5];
        /* YOU MUST ADD MATERIAL BELOW THIS LINE */




};

void driver(thread_id tid, Monitor* mon, int desired_car) {
     /* you should not modify this function */
     mon->take_key(desired_car);
     drive();
     mon->return_key(desired_car);
}

void Monitor::take_key(int desired_car) {
     /* YOU MUST FILL IN THIS FUNCTION. Note that the argument refers
        to the desired car. */









}

void Monitor::return_key(int desired_car) {
     /* YOU MUST FILL IN THIS FUNCTION. Note that the argument refers
        to the desired car. */









}
```

```
/* easiest way to extend monitor is with just a single condition
variable (could also have one condition variable for each key) */

class Monitor {

   .....

   Cond cond;
};

void Monitor::take_key(int desired_car)
{
    int which_key = desired_car / 2;

    acquire(&mutex);

    while (keys[which_key] != FREE) {
        wait(&mutex, &cond);
    }

    keys[which_key] = IN_USE;
    release(&mutex);
}


void Monitor::return_key(int desired_car)
{
    int which_key = desired_car / 2;
    acquire(&mutex);

    keys[which_key] = FREE;
    cond_broadcast(&mutex, &cond);
    release(&mutex);
}
```

II Virtual memory and paging (13 points total)

4. [2 points] Circle True or False:

True / False A virtual memory system that uses paging is vulnerable to external fragmentation.

False. Paging provides fine-grained allocation of memory, so the only fragmentation that can happen is wastage within a page, which is internal fragmentation.

5. [9 points] Consider a 32-bit computer with the following funky virtual memory architecture:

- Each page is 2KB (2^{11} bytes).
- Physical memory is 32 GB (2^{35} bytes). Note: that is not a typo; we do not mean 2^{32} bytes.
- Associated with each virtual page are 7 bits in the page table, including control and reserved bits. The exact nature of these bits is unimportant for this question. For completeness, they are: three bits controlled by the kernel (`PTE_P`, `PTE_U`, and `PTE_W`); 2 bits controlled by the hardware (Accessed and Dirty); and 2 bits reserved for the kernel (for example, `PTE_COW`).
- The machine has a two-level page table structure analogous to the one used in JOS: each process has a page directory, each entry of which points to a page table. Each entry in the page directory has the same number of control and reserved bits as a page table entry.

Below, state the minimum number of bits in a second-level page table entry, and explain *briefly*:

31 bits. There are 2^{35} bytes of memory and 2^{11} bytes/page, which means there are 2^{24} pages in the machine. That means that the second-level page table has to have at least 24 bits (to be able to select the physical page). Then there are $3+2+2=7$ further bits per page, so we need at least $24+7=31$ bits per entry.

Below, state the minimum number of bits in a page directory entry, and explain *briefly*:

31 bits, again. The reason is the same. The page directory entries need to be able to "resolve" 2^{24} bits since a page table could conceivably live on any physical page. Further, we are given that the number of control and reserved bits is the same in a page directory entry.

Now assume that the entry size is rounded up to the nearest multiple of 4 bytes. Further assume that a page directory or page table must fit on a page. **Programs on this machine use 32-bit quantities as instruction operands, but when the operand is an address, not all of these 32 bits are examined by the processor. How many address bits are actually used in this architecture?**

29 bits. A page table entry is 4 bytes (per the padding), and a page table is 2KB (because it needs to fit on a page). Thus, a page table can have no more than 512 (2^9) entries, which means that no more than 9 bits are used to index into the page table. Same for the page directory. Finally, since a page is 2KB, 11 bits of the address determine the offset into the page. Altogether, this is a maximum of $9+9+11=29$ bits of address in use.

How large is the *per-process* virtual memory space?

512 MB only. That is because a process can use only 29 bits of addresses, and $2^{29}=512$ MB. Note that on modern 32-bit x86s, something similar happens: each process gets 2^{32} bytes of virtual address space, but the machine can actually have more physical memory than that.

6. [2 points] Consider a machine with 32 MB of RAM running an operating system with virtual memory and swapping. The OS's page replacement policy is: if, on a page fault, a process needs a new physical page in RAM, evict the page that has been in RAM in the *longest*, and write it to the disk if it is dirty. The machine owner notices that for some workloads, the operating system does a lot of disk writes, and the owner is unhappy about that. In response, the owner installs an extra 8 MB of RAM, and re-runs the workload.

Circle True or False for each item below:

True / False There are workloads for which the extra RAM will *decrease* the number of page faults.

True / False There are workloads for which the extra RAM will *have no effect on* the number of page faults.

True / False There are workloads for which the extra RAM will *increase* the number of page faults.

This question was on the midterm too. All are true. For the first item: the working set might fit into 40 MB but not 32 MB of RAM. In that case, the extra RAM will decrease the number of page faults. For the second item: the workload might be pathological, say looping through all of its memory. Because this workload exhibits no locality of reference, the FIFO cache will not help: *every* reference could generate a page fault, whether there are 32 MB or 40 MB of RAM (in fact there are pathological workloads for which the gains of *any* cache, not just a FIFO-managed one, will be miniscule). For the third item, consider *Belady's anomaly*, which we discussed in class and which was covered again in the homework: for some access patterns and a FIFO replacement policy, increasing the size of a cache may *increase* the number of cache misses.

III Disks, file systems, I/O, transactions (15 points total)

7. [8 points] Consider a disk with the following characteristics:

- The disk rotates at 12,000 RPM (rotations per minute)
- The disk has 10 platters (and 10 corresponding heads); the cost to change which head is active is zero
- Each sector is 512 bytes
- There are 1024 sectors per track (we are ignoring the fact that the number of sectors per track varies on a real disk)
- There are 4096 tracks per platter
- The track-to-track seek time is 0 milliseconds
- If the disk head has to seek more than a single track, the seek time is given by $1 + t \cdot .003$ milliseconds, where t is the number of tracks that the disk is seeking over.
- Ignore the time to transfer the bits from the disk to memory; that is, once the disk head is positioned over the sector, the transfer happens instantaneously.

What is the storage capacity of the disk in bytes or gigabytes? Explain *briefly* (for example by showing your work).

Answer: 20 GB/disk. 10 platters/disk * 4096 tracks/platter * 1024 sectors/track * 512 bytes/sector = 10*4*1024*1024*512 = 40*1MB*512 = 40*.5GB = 20 GB.

What is the sequential transfer bandwidth, expressed in bytes/second or megabytes/second? Explain *briefly* (for example by showing your work).

Answer: 100MB/second. First note that 12,000 RPM = 200 rotations per second (or one rotation per 5 ms). In a single rotation, we can read an entire track. A track consists of 512 bytes/sector * 1024 sectors = 0.5 MB. So the sequential transfer bandwidth is 200 rotations/second * 0.5 MB/rotation = 100 MB/second. Because the track-to-track seek time and the I/O bus overhead are both modeled as negligible, 100 MB/second is our answer.

Name: Solutions UT EID:

Now assume that the disk with the above characteristics is given a never-ending stream of requests to read one sector at a time, with each request chosen randomly from all possible sectors on the disk. Assume that these read requests are scheduled in FIFO order. **In the space below, state the effective long-term transfer rate that the disk can sustain, expressed in bytes/second or kilobytes/second, and explain briefly.** The following may be useful:

- You can (and probably should) make several percentage point approximations, for example 4096 can be represented as 4000, and $13 \cdot 7.5 \approx 100$.
- If we pick a track uniformly at random and then pick another track uniformly at random, the expected (or average) number of tracks between them is *one-third* (not one-half) the total number of tracks on a platter (this can be derived using probability theory).
- What does the term "long-term transfer rate" actually mean? Here is a precise definition, plus a hint about how to calculate it. If this confuses you, then just ignore it and follow your intuition about what the term means, since what we're doing here is formalizing the usual intuition. Let $N(t)$ be the number of disk reads that take place over a time interval $[0,t]$. Define the *long-term read rate* as $\lim_{t\to\infty} \frac{N(t)}{t}$; observe that the units of this quantity are reads per second. We define the *long-term transfer rate* as $\lim_{t\to\infty} R \cdot \frac{N(t)}{t}$, where R is the number of bytes transferred in each read. **Now a hint:** A very useful fact from probability theory is that $\lim_{t\to\infty} \frac{N(t)}{t} = \frac{1}{X}$, where X is the average length of time that a read takes (this is the strong law of large numbers for renewal processes). Thus, the long-term transfer rate is $\frac{R}{X}$.

Answer: roughly 67 kbits/second. First note that in one read, we get 512 bytes. What is the time to issue this read? The disk incurs seek delay and rotational delay. The average seek latency is $1 + (1/3).003 * 4000 = 5$ ms. After the disk head reaches the desired track, the disk has to wait until the desired sector rotates under the disk head. Since the sector could be anywhere on the track, ranging from right under the head to the most pessimal position, the average rotational delay is 2.5 ms (half of the 5 ms per rotation). So the total delay on average is 7.5 ms. Our total effective bandwidth, then, is 512 bytes / 7.5 ms ≈ 512*130 bytes/second = 66,560 bits/second = 66.6 kbits/sec. Using the hint, $R = 512$, $X = 7.5$ms, etc.

8. [3 points] We discussed the Unix FFS (Fast File System) as an improvement over the classic Unix file system, which had serious performance problems. Which of the following techniques does FFS use?

Circle all that apply:

A Using DMA transfers to get data from the disk directly into memory

B Clustering related objects together in cylinder groups

C Setting inodes to copy-on-write to make directory changes fast

D Tracking free blocks with an in-memory bitmap

E Telling users that the disk is full when it isn't

F Implementing large directories as B+-trees for faster lookups

B,D,E. A surely helps performance, but this change happens below the layer of the file system; whether data is DMAed is up to the hardware and the device driver and is not something that the file system can control. C is not intended to be sensible in this context. F is implemented in some file systems but not FFS; recall that FFS used almost the same data structures as the traditional Unix file system.

9. [2 points] Recall the undo/redo logging protocol that we saw in class that was used to help implement transactions. The protocol worked with a log and some on-disk data structures (such as a database's tables, as implemented by B+-trees). These on-disk data structures are also known as *stable storage*. There were two invariants: (1) we always wrote a CHANGE record to the log before applying that change to stable storage; and (2) we only wrote the END record after fully applying the transaction to stable storage (or fully aborting it). The transaction's OUTCOME record (COMMIT or ABORT) was written at some time after the transaction's last CHANGE record and before the transaction's END record.

True / False By restricting the order in which the OUTCOME record is logged versus when the transaction's changes are installed in stable storage, we can ensure that the recovery process only ever needs to perform undos, never redos.

True / False By restricting the order in which the OUTCOME record is logged versus when the transaction's changes are installed in stable storage, we can ensure that the recovery process only ever needs to perform redos, never undos.

Both are true. As covered in the l21 notes, if all installs to stable storage happen before the relevant OUTCOME record is logged, then redos are never necessary. Likewise, if all installs to stable storage happen only after the relevant OUTCOME record is logged, then undos are never needed during recovery.

10. [2 points] The purpose of two-phase locking within transactions is:

Circle the best answer below:

- **A** To make the results of all transactions correspond to a serial execution of all of the transactions
- **B** To ensure that, after a crash, each transaction has either happened in its entirety or not at all
- **C** To allow a group of machines to all agree to perform, or not perform, their respective pieces of a distributed transaction
- **D** To avoid deadlocks
- **E** To prevent gas and liquid from escaping

A. Recall that the purpose of two-phase locking is to achieve before-or-after atomicity. B. is wrong because what ensures all-or-nothing atomicity is the logging and crash recovery protocols. C. is wrong because it refers to two-phase commit, not two-phase locking. D. is wrong because two-phase locking actually introduces a deadlock problem that is handled with another mechanism.

[Credit Saltzer-Kaashoek: *Principles of Computer System Design: An Introduction*, Exercises at the end of Chapter 9: Atomicity: All-or-Nothing and Before-or-After.]

IV Networks, NFS, RPC, distributed systems (19 points total)

11. [2 points] Which of the following systems experienced congestion collapse as a result of fixed retransmission timers? The Saltzer-Kaashoek readings will be helpful here; in fact, you can assume that if an item was not discussed in that reading assignment, then it does not need to be circled.

Circle all that apply:

A Early versions of Ethernet

B Early versions of NFS

C An outage at Google News, when Michael Jackson died

D The Therac-25

E A time server in Wisconsin

B and E, per the Saltzer-Kaashoek reading. For A, Ethernet was designed to avoid fixed timers. C was not discussed in Saltzer/Kaashoek, and the commentary on this subject seems to implicate a lot of human-driven traffic. The phenomenon of humans' continually pressing Refresh on their Web browsers is not the same thing as a fixed retransmission timer, though the resulting traffic patterns may be very similar!

12. [3 points] Two computers, M and N, are connected by a cross-country network link. The link runs at 10 megabits per second ($10 \cdot 10^6$ bits/second) in both directions simultaneously. The propagation delay is 24 milliseconds: that is, it takes 24 milliseconds for a bit to propagate from M to N or from N to M. M and N send fixed-size frames of 10,000 bits (1250 bytes). M and N run a ping-pong protocol: M sends a frame, then N replies with a frame, and only then does M send another frame. Assume that M sends a frame right after getting N's reply (but never before). Further assume that processing time at the two endpoints is zero. Last, assume that the link never drops frames.

What is the maximum number of bits per second, taken over the long-term, that M can send to N using the above link and protocol?

To send 10,000 bits on a 10 Mbit/s link takes 1 millisecond (ms). It further takes 24 ms to send a single bit, so it takes 25 ms to send a frame, and, since processing costs are ignored, another 25 ms to receive one back. Thus, M can send one frame, which is 10,000 bits, every 50 ms. This data rate = 10,000 bits/50 ms = 200 kbits/second.

The next question concerns the *correct* answer to the above question, so you do not need to answer the question above to answer this question. Relative to the theoretical line rate of 10 megabits per second, the *correct* answer to the above question is:

Circle the best answer below:

A Larger by orders of magnitude

B Slightly larger

C Roughly the same

D Slightly smaller

E Smaller by orders of magnitude

E, because 200 Kbits/s is two orders of magnitude smaller than 10 Mbits/s.

13. [4 points] Which of the following statements about Ethernet is true?

Circle all that apply:

A Currently deployed Ethernets have a maximum physical range

B Currently deployed Ethernets have a minimum frame size

C Every Ethernet device in every personal computer sold has a unique address configured

D If a sender and receiver are not connected to the same Ethernet hub, bridge, or switch (that is, they are on different local networks), then the receiver's Ethernet address provides crucial information to help the sender's network direct the sender's packets to the receiver's network

E People use switches to connect Ethernet networks to each other

F The Internet is a collection of connected Ethernet switches

G When a sender detects repeated collisions, it uses exponential backoff to select a random time to retransmit

A,B,C,E, and G are all true. (However, we gave credit for either answer for E because some people defined "Ethernet network" to mean "a collection of local switches" instead of "literally shared extent".) D is false because, if the sender and receiver are not part of the same Ethernet, then the receiver's Ethernet address not only isn't useful for the sender but also is unknown to the sender. F is false because while the Internet has many connected Ethernet switches, it has much more besides. In particular, many Ethernet networks connect to each other via routers. And people connect to the Internet with satellite links and other non-Ethernet link layers.

14. [3 points] Which of the following are RPCs supported by an NFS server, as reported in "Design and Implementation of the Sun Network File System"?

Circle all that apply:

A `fh = LOOKUP(directory_filehandle, file_name)`

B `fh = OPEN(file_path_and_name, mode)`

C `rc = CLOSE(filehandle)`

D `attr = WRITE(filehandle, offset, num_bytes, data)`

E `attr = WRITE(filehandle, num_bytes, data)`

F `trans_id = BEGIN_TRANSACTION(filehandle, offset, num_bytes, data)`

G `outcome = COMMIT(trans_id)`

Only A and D. The others imply state at the NFS server, and the NFS server reported in the paper is stateless.

15. [2 points] As discussed in class, an NFS file handle consists of three parts. Below we list two of the parts and ask you to fill in the third:

```
[ filesystem ID | XXXXXXX | generation # ]
```

What does the `XXXXXXX` above stand for?

The XXXXXXX stands for the inode #. The NFS file server is stateless, so the file handle needs to contain all of the information needed for the file server to process the RPC.

16. [5 points] Jamie T. Coder is a graduate student in meteorology who has also studied some operating systems and distributed systems. Jamie's game plan is as follows. Jamie is going to program two computers, A and B, though not necessarily with the same program. Once the programs are ready, Jamie will spend a busy morning and early afternoon flying around, installing and powering on these two computers in their field locations, one in Houston and the other in Chicago. Each of these computers is supposed to log weather measurements (for example, the temperature and atmospheric pressure) in their respective locations at 4:45 PM each day. Note that Houston and Chicago are in the same time zone, and you may assume that the clocks in A and B are perfectly synchronized and remain so. Jamie's advisor imposes the following requirement: on any given day, either both computers should log their measurements at 4:45 PM, or neither should. The computers will be connected to the Internet (which of course can drop, reorder, duplicate, or delay network packets), and the programs that Jamie writes may send and receive messages over the Internet.

Below, note that instructions and questions are interleaved; please read carefully.

Assume that A and B are fully reliable. That is, Jamie is assured that they never crash or malfunction.

Circle True or False:

True / False Jamie can write code to run on A and B that guarantees that A and B both log their measurements at 4:45 PM each day.

True. Jamie doesn't need the network; Jamie simply programs A and B to perform the action at 4:45 PM each day.

Assume that A is fully reliable but B can sometimes crash, after which it always recovers.

Circle True or False:

True / False Jamie can write code to run on A and B that guarantees that, on any given day, A and B either both log their measurements at 4:45 PM, or both don't.

Credit for True, False, or blank. The intended answer was False. The fact that B can sometimes crash means that A cannot be programmed to always take the action (even if B can). This in turn means that A will have to detect if B is up or not. This in turn means that B has to expect that A does not know if B is up or not. Unfortunately, A cannot reliably detect whether B is up because the network can drop messages, and B cannot be assured that A knows it is up. So we are back to the two generals' problem, which is impossible to solve.

However, we failed to specify a liveness requirement in the problem, which meant that code by Jamie that did nothing at all would satisfy the question as stated ("either both log or both don't"). Thus, we gave credit for any answer, including blank.

Assume that A is fully reliable but B can sometimes crash, after which it always recovers, and Jamie can use the services of a fully reliable machine, C, in the lab in Austin (that is, Jamie is assured that C never malfunctions or crashes).

Circle True or False:

True / False Jamie can use two-phase commit, with C acting as the coordinator, to guarantee that, on any given day, A and B either both log their measurements at 4:45 PM, or both don't.

Name: Solutions **UT EID:**

Credit for True, False, or blank. The intended answer was False. Two-phase commit can ensure that all computers eventually perform the action or not, but in the presence of an unreliable network and/or unreliable machines, two-phase commit cannot ensure that the action always happens by a specified deadline.

However, we failed to specify a liveness requirement in the problem, which meant that code by Jamie that did nothing at all would satisfy the question as stated (that is, Jamie could use two-phase commit, but with the 2PC having no interesting purpose). Thus, we gave credit for any answer, including blank.

V Protection, privilege, security, and setuid (16 points total)

17. [6 points] In this problem, you will help conduct a return-from-libc attack. Here is the backstory. After her account is hacked in CS372H, Namrata grows vengeful and turns to crime. She intently reads the source code of the Web server running at `www.cs.utexas.edu` and finds in that code a buffer overflow vulnerability. Unfortunately for Namrata, the OS marks the Web server's stack pages non-executable, so the attack that we saw in lecture won't work. However, Namrata remembers from Tanenbaum that a buggy server with a non-executable stack can still be exploited, using a *return-from-libc* attack. (If this attack does not sound familiar, this question will be a bit harder but definitely still doable.) Namrata figures that she can use a return-from-libc to get a shell that runs with the Web server's privileges (allowing her to deface any UTCS Web page, including the one for CS372H).

Owing to the Web server's bug, Namrata can write *anything* onto the server's stack, starting at the stack pointer (SP) depicted below. We depict the server's stack before and after Namrata's attack, but we have left out four key details, namely the values that go in stack memory addresses 0xeeb00100, 0xeeb00104, 0xeeb00108, and 0xeeb0010c. Your job is to state what these values should be by marking the figure. Here is some useful information about the server's address space:

- The code that Namrata is injecting `exec`s a shell. She wants this code to wind up at, and then to be run from, address 0x300000, which is a program text area that is both writable and executable.
- Namrata knows that `memcpy` is located at 0x700000 in the server's address space.
- For our purposes, `memcpy` is part of libc and is declared like this:
  `memcpy(void* dest_addr, void* src_addr, uint32_t size);`
  It copies `size` bytes from memory address `src_addr` to memory address `dest_addr`.

Fill in the values of V1, V2, V3, and V4 that will let Namrata `exec` a shell from the Web server:

18. [2 points] This question is about root, otherwise known as the superuser.

Circle True or False:

True / False A process running as root is permitted to execute privileged x86 instructions, for example, instructions to load the `cr3` register or clear interrupts.

False. The root user is a construction of the operating system. It transcends the *operating system's* privilege mechanism but not the *hardware's*. For example, the root user can bypass all file access checks because the operating system is the author of both the root user and the file access restrictions. However, the root user cannot execute operations that the hardware requires supervisor mode to execute: from the hardware's perspective, root is just another user.

19. [8 points] Consider a compiler called `cc` that runs on a machine that has many users. This compiler is a normal program—it is *not* setuid—and is invoked by a user as follows:

```
$ /bin/cc -o <outfile> <infile>
```

Above, `<infile>` is the input source file and `<outfile>` is the name of the compiled output program. Now, assume that the administrators of the machine want to instrument the compiler to deposit into a file called `/etc/stats` statistics about what kinds of files `cc` is compiling and how long those compilations take. To do so, they add some lines to the compiler's source code:

```
int stats_fd; /* declared as global variable */

int main(int argc, char** argv) {
    .....
    /* early on in main() */
    if ((stats_fd = open("/etc/stats", O_WRONLY|O_APPEND|O_CREAT)) < 0) {
          fprintf(stderr, "bummer, dude\n"); /* prints error message */
          exit(-1);
    }
    ....
}
```

Then, wherever in the code they want the compiler to log statistics, they add code like:

```
if (write(stats_fd, buffer, size) != size) {
      fprintf(stderr, "major problem, dude\n"); /* prints error message */
      exit(-1);
}
```

where `buffer` holds a line of statistics and `size` is the number of bytes in the buffer to dump in the stats file. The administrators then recompile `cc` itself and install it as `/bin/cc`.

Assume that the directory `/etc` is writable only by root and that the file `/etc/stats` is readable by all but writable only by root.

After instrumenting the compiler per the above, the administrators start getting a flood of complaints from their users, none of whom can get useful work out of the compiler. **Below, state in a few words what the user sees after typing `/bin/cc -o hello hello.c`, and also explain in a few words why the user sees this.**

The user sees "bummer, dude" printed to the screen, and the reason is that the compiler runs as the user, who does not have permission to write the `/etc/stats` file.

To address the above problem, the administrators of the machine make the compiler setuid root. That is, they set the setuid bit on `/bin/cc`, ensuring that when it runs, it does so with root's privileges. Their change addresses the problem just above: users are able to get useful work out of the compiler. However, there is a huge problem: an attacker can now overwrite, say, `/etc/password`. **Below, give an example command that an antisocial user could invoke to cause `/etc/password` to be destroyed, and explain why this works.**

The attacker can issue

```
$ /bin/cc -o /etc/password hello.c
```

This will work because the compiler has all of root's privileges, which include the ability to open and write to any file. This issue, by the way, is called "The Confused Deputy Problem", and we return to it in some depth in the graduate operating systems course.

The administrators try to address their error by hacking the source code for the compiler again. Without rolling back the prior changes, they add code to check the `<inputfile>` and `<outputfile>` arguments using the `access()` system call. `access()` checks whether the process's real user id (not its effective user id) can open the file with the given permissions. Here's an excerpt from their code:

```
int output_fd, input_fd;
/* output_file is extracted from the command line */
if (access(output_file, W_OK) < 0) {
      fprintf(stderr, "user should not open %s for writing\n", output_file);
      exit(-1);
}
if ((output_fd = open(output_file, O_WRONLY|O_TRUNC|O_CREAT)) < 0) {
      fprintf(stderr, "couldn't open %s for writing\n", output_file);
      exit(-1);
}
.....
```

Below, explain *briefly* the problem in the code above, and how an attacker could exploit it:

The issue is a TOCTTOU (time of check to time of use) error. The attacker could do the following:

```
$ /bin/cc -o /tmp/foo hello.c
```

Then, *as* that is running, the attacker gets the timing right so that after `cc` does the `access()` but before the compiler does the `open()`, the attacker manages to do:

$ rm /tmp/foo

$ ln -s /etc/password /tmp/foo

The result is that the `open()` line will clobber `/etc/password` because the compiler is running setuid.

VI Readings and guest lecture (17 points total)

20. [6 points] All of the items below concern LFS, the log-structured file system. Note that instructions and questions are interleaved below, so please read carefully.

Circle True or False for each of the two items below:

True / False A key motivation for the LFS is the expectation that disk traffic consists primarily of write requests.

True.

True / False Compared to an instance of Unix FFS (Unix Fast File System) that does not use journaling, LFS permits faster crash recovery.

True. After a crash, LFS needs to scan only the most recent part of the log, not the whole disk, as on a standard Unix file system.

Recall that the log is divided into segments. Further recall that a key problem is garbage collection: making room for new segments. Finally, recall that the method of garbage collection is to *clean* segments, which means to gather the currently used parts of several segments and compress them into a single segment. A key question is when a segment should be cleaned (that is, compressed).

Circle True or False for each of the two items below:

True / False A segment in LFS can only become a candidate for cleaning when its utilization drops below k_u, where k_u is a constant given in the LFS paper.

True / False A segment in LFS is *always* cleaned when its utilization drops below k_N, where k_N is a constant given in the LFS paper.

Both of the above are false. k_u and k_N do not exist. LFS's decision to clean a segment is a function of the segment's utilization and its age, not hard-coded constants. Specifically, as discussed in class, LFS picks the segment with the highest cost-benefit ratio, calculated as $\frac{(1-u)\cdot a}{1+u}$, where a is age, and u is utilization. Thus, there is no set utilization below which all segments are cleaned or above which a segment is never cleaned.

This question turned out to be trickier than we intended. The reason is that there *is* a constant threshold in the paper that relates to cleaning. However, that threshold is not what this T/F was asking about. That threshold concerns when the cleaning process overall is invoked and has nothing to do with the utilization of individual segments; for the purposes of that threshold, all segments with utilization greater than zero are equivalent.

For the items below, assume that a 100 megabyte file already exists on disk and that a benchmark program writes to the file at randomly chosen locations ("random writes") and then re-reads the file sequentially ("sequential reread"). The benchmark program is run on two computers that are identical, except that one has LFS and the other has Unix FFS.

Circle True or False for each of the two items below:

True / False The LFS computer completes the random writes in less time than the FFS computer.

True / False The LFS computer completes the sequential rereads in less time than the FFS computer.

The first is true, and the second is false. Figure 9 in the LFS paper makes this clear. Any writes, including random access writes, file creation, file deletion, etc. are fast in LFS. Sequential rereads are slower in LFS because the reread, though *logically* sequential under LFS, is actually not going to involve reading the disk sequentially. Under FFS, the reread will be mostly sequential since FFS succeeds in placing files mostly sequentially.

21. [2 points] To demonstrate steganography, Tanenbaum includes two identical-looking photographs, one a standard photograph and the other encoded with additional information that the human eye cannot detect. What is that additional information?

According to Tanenbaum in his book, he included the complete text of five plays by William Shakespeare: *Hamlet*, *King Lear*, *Macbeth*, *The Merchant of Venice*, and *Julius Caesar*.

22. [2 points] Tanenbaum includes a section on rootkits. Which of the following rootkits does he describe in depth?

Circle the best answer:

- **A** The one installed by the hack to the C compiler that left no trace in the source
- **B** The one installed by the hacking of `cron` to make it execute the attacker's instructions
- **C** The one installed by the first Internet worm
- **D** The one installed by Sony BMG
- **E** The one that created a hypervisor, ran the entire operating system inside the hypervisor (prohibiting any program that ran inside the OS from finding the rootkit, thanks to hypervisor-based isolation), and caused $50,000,000 worth of damage worldwide before being discovered.

D. The others are made-up incidents, though E could very well have happened. To our knowledge, Thompson did not install a rootkit (ruling out A), the hacking of cron was described but not in the context of a rootkit (ruling out B), and the Internet worm did not hide its presence (ruling out C). Tanenbaum describes the Sony BMG rootkit at length.

23. [2 points] Who wrote, "There is an old adage: 'Dance with the one that brought you,' which means that I should talk about UNIX." ?

Ken Thompson wrote it, in "Reflections on trusting trust".

24. [3 points] In lecture, Keith Winstein said several times that many computers sold today are *not* Turing Machines. He did not mean that Turing Machines have infinite memory whereas computers sold today have finite memory. So what did he mean? If you know what a Turing Machine is, skip the description that follows and just answer the boldface question below. If you could use a review of Turing Machines, here is a brief description that gives all the background needed to answer this question. A Turing Machine is a theoretical model of a computer that has an infinite, linear tape. The Turing Machine reads instructions from this tape and moves around on the tape, reading from it and writing to it. For the most powerful Turing Machines (the kind that Keith was talking about), the "programmer" can, in theory, cause the machine to perform *any* computation via suitable choice of the starting set of instructions on the tape. Note that these instructions model a computer program. (As an aside, because the basic model of a Turing Machine uses a linear tape, in contrast to computers' RAM, it performs its computations inefficiently. However, this aside is totally irrelevant to this question.)

Explain briefly below what Keith meant when he said that many computers sold today are not Turing Machines.

Turing Machines can perform arbitrary (unconstrained) computations. Many computers sold today will not perform arbitrary computations. Owing to some of the technologies that Keith mentioned, these computers will run only programs that the manufacturer blesses, or that have a required key. (In literal terms, Keith was not correct: the Turing Machine formalism covers very weak models of computation, including deterministic finite automata. But the question asked what he meant.)

25. [2 points] Keith said that the most important thing when designing a solution that is supposed to be secure is:

Select one:

A Ensuring that your data is encrypted

B Making your algorithm public

C Specifying your threat model

D Ensuring that you're not inventing ad-hoc cryptography

E Outsmarting Jack Valenti

Answer: C. Keith said, rightly, that a threat model is crucial to define so that you know exactly what your system is defending against. Answer A may not be necessary (depends on the threat model and the details of the situation). Answers B and D are very important too.

VII Labs and feedback (8 points total)

26. [5 points] Your lab 6 network card driver uses a DMA ring to store network packets generated by user-level processes. The transmit DMA ring is a circular linked list (the links being physical addresses). Recall the format of an entry in the transmit ring:

```
struct tcb {
 volatile uint16_t status;
 uint16_t cmd;
 uint32_t link; /* physical address of next tcb in ring */
 .....
 uint16_t byte_count; /* number of useful bytes in 'data' field below */
 ....
 char data[MAX_ETH_FRAME]; /* the data that will be DMAed by the card */
};
```

Soon after boot, the driver configures the card to be aware of the head of this list. As a result, the card can now transfer data from host memory to itself. But now we have a classic producer-consumer pattern: the host is writing to a circular buffer, and the card is reading from that buffer. Indeed, absent judicious design, we could have race conditions and liveness violations. Example races are the card's reading data before the host has finished writing to the TCB in question, the card's reading old data from the host, and the host's overwriting data before the card has consumed it. An example liveness violation is if the card never realizes that the host has new data to send.

Briefly **state the high-level mechanism(s) that the card and your device driver use to avoid race conditions; high-level means that you need not name exact bits, registers, or electrical signals:**

Briefly **state the high-level mechanism(s) that the card and your device driver use to ensure liveness:**

27. [2 points] This question is about the division of responsibility on your final project. If you had no partner for the final project, write "No partner" below. Credible answers will receive full credit. *A blank answer will earn zero points.*

Please list the components in the final project that your partner wrote most of the code for:

Please list the components in the final project that you wrote most of the code for:

28. [1 points] This is to gather useful feedback while we have your attention. For both of the questions below, any answer will receive full credit. *A blank answer will earn zero points.*

Please list the two topics in this course that you most enjoyed:

There is no right answer.

Please list the two topics in this course that you least enjoyed:

There is no right answer.

End of Final

Congratulations on having finished CS372H!

Enjoy the summer!!