The University of Texas at Austin
CS 372H Introduction to Operating Systems: Honors: Spring 2012
Final Exam

- This exam is **3 hours**. Stop writing when "time" is called. *You must turn in your exam; we will not collect it.* 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 **24** questions in this booklet. 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, calculators, laptops, 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 55 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 that you can answer crisply. Be neat. If we can't understand your answer, we can't give you credit!
- There is no credit for leaving questions blank. However, to discourage unfocused responses, we will be grading the clarity of your answer. Moreover, some questions impose "sentence limits".
- 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/33)	II (xx/18)	III (xx/29)	IV (xx/20)	Total (xx/100)

Name: Solutions **UT EID:**

I Unix, concurrency (33 points total)

1. [3 points] Consider the following statement: "On the x86, if a given memory reference (load or store) causes a page fault exception, then that memory reference also caused a TLB miss."

Is the above statement True or False? Justify your answer *briefly* below. You do not need more than two sentences.

False. The page fault could have resulted because a user-level process attempted to write to a page that was write-protected. The TLB contains this permission information (so there was no TLB miss), which is in fact what leads to the page fault.

2. [4 points] On Unix, a user-level program issues the following system call:

```
write(fd, buf, 10);
```

The file descriptor, `fd`, can refer to which of the following?

Circle ALL that apply:

- **A** A device
- **B** A file
- **C** A directory
- **D** A pipe
- **E** A network socket
- **F** A scatter-gather descriptor

All but C, F.

3. [4 points] On Linux, which of the following produces output in the ELF format?

Circle ALL that apply:

- **A** a command-line invocation of `$ gcc -S foo.c` # this produces foo.s
- **B** a command-line invocation of `$ as foo.s` # as is the assembler
- **C** a command-line invocation of `$ gdb foo`
- **D** the default linker
- **E** the default loader

B. and D.

Name: Solutions **UT EID:**

4. [3 points] Several times this semester, we have said that Unix's separation of the `fork()` and `exec()` system calls is powerful. This question asks you to give an example of how the programmer can use that separation.

Give an example that we have seen this semester of functionality that the programmer might insert between `fork()` and `exec()`. Be brief; the question is not asking you to write code.

5. [6 points] Pat Q. Hacker gets a job building a kernel for multiple-CPU machines. Pat implements a standard spinlock for use inside the kernel, because the kernel may be simultaneously executing on two CPUs and may need to synchronize access to global data structures. The spinlock exposes an API of `acquire()` and `release()`. Pat's implementation of `acquire()`, among other things, disables interrupts on the local processor; `release()`, among other things, reenables them.

Pat's kernel shares data between interrupt handlers and other kernel code; all of this code acquires and releases the spinlock. Assume that when a CPU is executing inside an interrupt handler, interrupts are disabled on that CPU (because interrupts vector through interrupt gates, as in JOS).

Now, Pat's manager, who dislikes disabling interrupts, argues that the code that executes between `acquire()` and `release()` need not run with the given CPU's interrupts disabled.

Pat's manager is incorrect; the code protected by spinlocks must run with interrupts disabled. Explain why *briefly* below.

The spinlock is shared between interrupt handlers and top-half ("regular") kernel code. If a processor has acquired the spinlock from top-half kernel code and then gets an interrupt that tries to acquire the spinlock, the processor will spin forever in the interrupt, trying, but failing, to acquire the spinlock.

If acquiring a spinlock disables interrupts, why do we need the spinning and the flag in the spinlock? Why not implement `acquire()` as simply "disable interrupts" and `release()` as simply "enable interrupts"? Explain briefly.

We need spinlocks to protect the shared data from access by an interrupt handler on *another* CPU (call it CPU B). Note that it's okay if the interrupt handler on CPU B spins while the locking CPU (call it CPU A) has the lock. The reason is that CPU B's spinning is bounded; it stops after CPU A releases the lock.

6. [4 points] We mentioned in class that, when writing code with multiple locks, the programmer must establish an order on locks and then acquire locks in that order. However, this approach is a manual process, which raises a question: why don't we build a compiler that complains when the programmer writes code that would violate the lock order?

Below, explain *briefly* (one or two sentences) why the compiler cannot enforce the lock order in all cases, even if the programmer declares the intended lock order to the compiler.

Because detecting a bug like "didn't follow the lock order" concerns detecting run-time behavior a priori, a task that is impossible in full generality (as is any decision problem of the form, "will program X perform action Y"?). Put differently, if the compiler could always detect, for all programs, that a program would violate a lock order, the compiler could solve the halting problem, which is undecidable.

7. [3 points] Imagine two scenarios, A and B. In scenario A, a two-processor machine provides sequential consistency, and two different threads run, one on each processor. In scenario B, the machine is a single CPU (so we don't need to talk about consistency), and the same two threads execute on the processor. Assume that the compiler does not reorder instructions. Define the *end state* of the computation as the state of memory after the computation terminates.

Consider the following statement: "The set of possible *end states* in each scenario is identical".

Is the above statement True or False? Justify your answer *briefly* below. You do not need more than two sentences.

True. Sequential consistency implies that, even with separate processors, one can write down an execution order that's consistent with the threads' having executed on a single processor, in program order. (Another way to understand this question is that without sequential consistency, shared-memory multiprogramming is even "worse" because executions can result in the multiprocessor case that could never have happened in the single processor case.)

8. [3 points] In the context of the Determinator reading ("Efficient System-Enforced Deterministic Parallelism", Aviram et al., OSDI 2010), we drew a distinction in class between *deterministic* execution and *predictable* execution (the latter being stronger than the former).

Describe this distinction in *one or two* sentences. We will read only the first two sentences.

Deterministic means that the process's execution, specifically the interleavings of its threads, is always the same, given the same inputs. Predictable is stronger: it is deterministic, and the programmer can guess how the program will execute.

9. [3 points] Recall that our treatment of Determinator included a guest presentation by Chris Cotter. Which of the following were mentioned by Chris in his presentation?

Circle ALL that apply:

- **A** Parent-child relationships in Linux's process model
- **B** Kernel-level synchronization primitives in Linux
- **C** User-level synchronization primitives in Linux
- **D** Memory-mapping hardware into his root process
- **E** A sample Web server
- **F** A sample matrix multiplication computation

A, B, F.

II Files, transactions, I/O (18 points total)

10. [4 points] Asssume that the current directory contains a file, `a`, and two directories, `b` and `c`. Invoking `ls` produces the following output:

```
$ ls
a    b    c
```

In contrast, invoking `ls -F` gives more information; in this example, it indicates to the user which of the entries is a directory:

```
$ ls -F
a    b/    c/
```

Below, explain *precisely* why `ls -F` causes the file system implementation to do more work, compared to `ls`. Use a maximum of two sentences.

To respond to ls, the file system simply has to find the directory contents and display them. To respond to ls -F, the file system has to fetch the referenced inodes, to learn which are directories, which are symlinks, etc.

11. [4 points] Recall that the log-structured file system (LFS, as in "The design and implementation of a log-structured file system", Rosenblum & Ousterhout, ACM TOCS, 1992) does not perform better than a traditional file system on all workloads. Consider the following workload. A benchmark program has a burn-in phase in which it starts with a fresh file system and then creates 1000 empty directories (this step does not count in the workload). The benchmark program then measures the performance of 10,000 creates and deletes of 1 KB files, with the enclosing directory chosen randomly from all 1000 directories in the file system. Is this workload one for which you would expect LFS or a traditional (optimized) Unix file system to do better? Assume that LFS's cleaning overhead is negligible.

Below, state which type of file system—log-structured or traditional Unix—you would expect to perform better on the above workload. Justify your answer *briefly*.

We expect LFS to do better because these file creates and deletes are "writes" to the files and to the relevant directories. That is, this workload is write-dominated, which is the kind of workload on which LFS excels. See Figure 8 in the LFS paper.

Name: Solutions UT EID:

12. [6 points] In class, we saw that a key component for implementing transactions with atomicity (which we sometimes called "all-or-nothing atomicity") was a logging protocol plus a recovery protocol. This question will ask you to adjust the logging protocol.

Recall that the system assumes (a) an on-disk log and (b) separate on-disk data structures (such as a database's tables), known as *stable storage*. The log protocol includes several record types, all of which include a transaction identifier:

- BEGIN: indicates the start of a transaction.
- CHANGE (also called UPDATE): logs a particular change that the system wishes to make to stable storage. Includes an "undo" action (which, if applied, undoes the change) and a "redo" action (which is the requested change).
- OUTCOME: includes whether the result of the transaction is COMMIT or ABORT. Expresses to the recovery protocol (which reads the log) what the actual result of the transaction was.
- END: indicates that the transaction is "complete", in the sense given below.

The logging protocol enforces two invariants:

- The system writes a change to stable storage only after logging the corresponding CHANGE record. (Here "logging" means "putting the log entry on disk".)
- The system writes the END record only after the transaction is fully applied to stable storage (or fully aborted in the sense that no trace of the transaction's updates appears in stable storage).

Finally, recall that recovery after a crash is a two-pass process: the system scans *backwards* from the end of the log, *undoing* some changes, and then scans *forwards*, *redoing* some changes.

What modification or addition to the invariants above could we make so that recovery could consist of only a single *undo* pass? *Be brief.*

Add an invariant that the OUTCOME record is written to disk only after all changes have been made in stable storage. At that point, there is no need to redo transactions that reached their commit point but somehow didn't make it to stable storage—because there are no such transactions. Recovery, then, consists only of undoing actions for which there is no OUTCOME record.

What modification or addition to the invariants above could we make so that recovery could consist of only a single *redo* pass? *Be brief.*

Add an invariant that the system begins propagating changes to stable storage only after the OUTCOME record is logged. At that point, there is no need to undo partially-applied transactions since there are no such transactions. Recovery consists of applying the actions that committed but whose transactions do not have an END record logged.

13. [4 points] This question concerns "Operating System Transactions" (D. E. Porter et al., SOSP 2009).

Circle True or False for each item below:

True / False TxOS provides transactional support for all system calls in Linux.

True / False TxOS provides transactional support for operating system state, not application state (heap, stack, etc.).

True / False If TxOS detects a conflict between two transactions, it chooses as the winner of the conflict the process with the higher number of shares.

True / False A technique that appears in TxOS is using shadow objects that correspond to key OS data structures.

True / False A technique that appears in TxOS is splitting key OS data structures into two pieces.

True / False A technique that appears in TxOS is efficient de-duplication of application-supplied buffers.

True / False A technique that appears in TxOS is special-case handling of linked lists.

False, True, False, True, True, False, True.

III I can't think of a unifying topic for this section (29 points total)

14. [4 points] This question concerns the reading "Design philosophy of the DARPA Internet protocols" (D. D. Clark, SIGCOMM 1988). Clark mentions various goals held by the original Internet architects (in a priority order). Which of the following goals does Clark NOT mention?

Circle ALL that apply:

- **A** Survivability in the face of failure
- **B** Support for heterogeneous network technologies
- **C** Support for heterogeneous network traffic
- **D** Security
- **E** Accountability
- **F** Compatibility with government regulation

D. and F.

15. [4 points] This question concerns the virtualization of virtual memory, as described in class and outlined in "Virtual Memory Management in VMWare ESX Server" (C. Waldsburger, OSDI 2002). Recall that *shadow page tables* are maintained by the virtual machine monitor (VMM) and map the guest process's virtual memory pages to machine pages (by *guest process*, we mean a process run by the guest OS). A *primary page table*, on the other hand, is maintained by the guest OS and maps a guest process's virtual pages to physical pages inside the virtual guest. (There is also a per-guest *pmap*, maintained by the VMM, that maps physical pages to machine pages or disk locations.) Both the shadow page tables and the primary page tables have the x86-defined page table structure.

For each of the four questions below, indicate by letter which entity performs the actions (choose the best answer). The choices are given after the questions.

Which entity writes the P (present) bit in the entries of the primary page tables? __________

Which entity writes the A (accessed) bit in the entries of the primary page tables? __________

Which entity writes the W (writable) bit in the entries of the shadow page tables? __________

Which entity writes the D (dirty) bit in the entries of the shadow page tables? __________

The choices are:

- **A** The guest OS
- **B** The guest process (running on the guest OS)
- **C** The virtual machine monitor (VMM)
- **D** The actual processor (on which the VMM runs)
- **E** Mike Dahlin

Name: Solutions **UT EID:**

A, C, C, D.

16. [6 points] Many scheduling algorithms are parameterized. For instance, the round-robin algorithm requires a parameter to indicate the time quantum. The multi-level feedback (MLF) queue scheduling algorithm has parameters to define the number of queues, the scheduling algorithm for each queue, the criteria to move processes between queues, and the criteria to interrupt a running process. Hence, each of these algorithms represents a set of behaviors. Further, one algorithm may simulate another in the sense that choosing its parameters appropriately may cause it to behave like another algorithm (for example, round-robin with infinite quantum duration is the same as first-come, first-served (FCFS)).

This question asks whether multilevel feedback queues and FCFS can simulate each other.

Can FCFS simulate multi-level feedback for all possible parameters of multi-level feedback? Justify your answer *briefly*.

No. FCFS runs jobs to completion in the order that they showed up, period. Multi-level feedback may implement, say, round-robin, in which processes alternate.

Can multi-level feedback simulate FCFS? Justify your answer *briefly*.

Yes. Choose one queue, and choose FCFS in that one queue.

17. [4 points] Consider a server with a buffer overflow vulnerability (for example, owing to a buggy server, a remote attacker can overwrite values on the server's stack). Recall that there were three components to the attack that we saw in class: (1) The attacker placed its own code on the stack (specifically code to `exec()` a shell), (2) The attacker overwrote the original return address in the server's stack frame; and (3) The replacement return address was that of the attacker-supplied code, causing the return from the function to set `%eip` to an address on the stack.

Now, as a defense to the attack above, say that the operating system marks a process's stack as non-executable (modern x86 hardware permits such marking via bits in the page tables). As a result, if the attacker attempts the attack above, then the program will crash once `%eip` points to the stack.

Consider the following statement, "Marking the stack non-executable is sufficient to avoid all exploits of buffer overflow bugs".

Is the above statement True or False? Justify your answer *briefly* below in three or fewer sentences.

The statement is false. As long as the attacker can control the return address—a case not ruled out by marking the stack non-executable—buffer overflow attacks are possible. Two examples are return-from-libc attacks, which were described in the textbook, and return-oriented programming, which we described in class. In neither case does `%eip` point to the stack but in both cases, the attacker subverts the flow of the program that the programmer had intended.

18. [5 points] Ken Thompson, in his lecture "Reflections on Trusting Trust" (Turing Award lecture, 1984), describes how he "extended" the C programming language. His hacked C compiler matched well-known strings in the source of `login` and in `cc.c`, which is the source of the C compiler itself. On matching these well-known strings, Thompson's hacked C compiler added bugs. Specifically, recall this figure from Thompson's paper:

```
void compile(char* s)
{
     if (match(s, "pattern1")) {
          compile("bug1");
          return;
     }

     if (match(s, "pattern2")) {
          compile("bug2");
          return;
     }
     ....
}
```

Thompson does not specify the language and format of `bug1` and `bug2`, but let us assume for this question that they themselves are expressed in C.

We saw that recompiling the compiler perpetuates the bug. But what happens if we modify the C compiler source to target a different hardware architecture? Does the attack persist, or does the new compiler start with a clean slate?

Specifically, imagine that we modify `cc.c` to create `cc-MIPS.c`, the source code for a compiler that transforms C code to MIPS binaries. Assume that `pattern2` occurs in `cc-MIPS.c`.

Now, if we compile `cc-MIPS.c` with the hacked compiler, we get a cross compiler, `cc-MIPS-cross`, that runs on our original architecture but produces MIPS binaries. If we then compile `cc-MIPS.c` with `cc-MIPS-cross`, we get `cc-MIPS`, a compiler that runs on the MIPS architecture and produces MIPS binaries. This question is asking whether `cc-MIPS` includes Thompson's bug.

Below, state whether the attack persists or not, and explain why. Your answer should be brief.

The attack indeed persists. `bug1` and `bug2` are themselves C programs. The second of these programs is self-reproducing: when compiled and executed, it logically inserts into the program being compiled some C statements that are equivalent to the excerpt from `compile()` above. This "logic" persists across all recompiles. This is actually quite disturbing: the hack is "portable" because the original self-replicating program was in the source. Moreover, even though the original attack was expressed in the source, the logic does not leave the binaries.

19. [2 points] From which of our assigned readings is this quotation? "Be *embarrassed* when someone else finds a bug in your code—you've wasted their time".

This is from MikeD's software engineering document.

Name: Solutions **UT EID:**

20. [4 points] In Tanenbaum's textbook, there is a figure in which he depicts two login screens: a correct one and a phony one.

Describe *briefly* how the phony one differs from the correct one.

It does not differ.

IV Labs, communication, feedback (20 points total)

21. [5 points] Consider the code below, from `kern/entry.S`. This question asks why instruction fetches do not generate page faults right after JOS turns on paging. The comments in the code give the context for this question. A summary of that context is immediately below.

When JOS begins executing, its code is located in physical memory at an address that is slightly higher than 1 MB. Also, paging is not enabled. Thus, the instruction pointer, `%eip`, is in the range [1MB, 2MB]. However, the kernel is linked to run at KERNBASE+1MB, meaning that the kernel mostly "expects" its code to live at KERNBASE+1MB (or slightly above). As a result, the initial page table translates virtual addresses in the range [KERNBASE, KERNBASE+4MB) to physical addresses in the range [0, 4MB). Recall that KERNBASE is 0xf0000000.

```
# We haven't set up virtual memory yet, so we're running from the physical address the
# boot loader loaded the kernel at: 1MB (plus a few bytes).  However, the C code is
# linked to run at KERNBASE+1MB.  Hence, we set up a trivial page directory that
# translates virtual addresses [KERNBASE, KERNBASE+4MB) to physical
# addresses [0, 4MB).  This 4MB region will be sufficient until we set up our real
# page table in mem_init in lab 2.

# Load the physical address of entry_pgdir into cr3.  entry_pgdir
# is defined in entrypgdir.c.
movl $(RELOC(entry_pgdir)), %eax
movl %eax, %cr3

# Turn on paging.
movl %cr0, %eax
orl $(CR0_PE|CR0_PG|CR0_WP), %eax
movl %eax, %cr0

# (*) Now paging is enabled, but the EIP is still low.
# Why is this okay?

# Jump up above KERNBASE before entering C code.
mov $relocated, %eax   # <--- WHY NO PAGE FAULT?
jmp *%eax

relocated:

# Clear the frame pointer register (EBP)
# so that once we get into debugging C code,
# stack backtraces will be terminated properly.
movl $0x0,%ebp # nuke frame pointer

# Set the stack pointer
movl $(bootstacktop),%esp

# now to C code
call i386_init
```

Name: Solutions **UT EID:**

Notice the comment and question marked `(*)`. You should answer that question. Specifically, why doesn't the instruction fetch of the line marked `<---` generate a page fault?

Give your answer below in one or two sentences. We will not read past the second sentence.

22. [5 points] Which challenge problem did you implement for lab 5? (Lab 5 was about the file system.)

State your answer in a sentence or two below. You do not need to describe your approach, only which challenge you selected.

23. [8 points] This question tests both your technical understanding, and your ability to communicate that understanding. Consider the following:

- Waldsburger's ballooning technique (in the VMWare ESX paper).
- Liedtke's technique (in "Improving IPC by kernel design", SOSP 1993) of direct transfers through temporary mappings.
- Jon Howell's use of a trampoline in his binary rewriting lecture.

Choose *one* of the three technical mechanisms above. For your choice, explain the technique: explain the "what" of it and the "why" of it. Imagine that your audience is a colleague who is technically sophisticated but unfamiliar with the technique in question.

Note that the words that you choose matter. It is not enough for your answer to convince us that *you* understand; your answer should also make someone *else* understand. Excessively long answers will count against you, so you will have to make choices about what to say and what not to say.

Describe one of the techniques above. Do not use more than a paragraph (roughly 5 sentences).

Here's an example answer, for ballooning. A VMM must sometimes decide which machine pages to reclaim from a given guest VM. In the ballooning technique, the VMM slyly gets the guest to make the decision. The technique works by installing a balloon device driver in the guest; the driver asks the guest OS to hand some *physical* page P to the driver. At that point, the VMM, which is in cahoots with the balloon driver, determines which *machine* page M corresponds to P, and reallocates M to another virtual machine.

24. [2 points] This is to gather useful feedback while we still 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.

Name: Solutions **UT EID:**