# The University of Texas at Austin
# CS 395T Operating Systems Implementation: Fall 2010
## Midterm Exam

- This exam is **75 minutes**. Stop writing when "time" is called. *You must turn in your exam; we will not collect them.* Do not get up or pack up between 70 and 75 minutes. The instructor will leave the room 78 minutes after the exam begins and will not accept exams outside the room.
- There are **14** 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 notes. You may not use electronics: phones, PDAs, calculators, etc.** You may use your copies of the papers—but only if they do not have class notes on them.
- 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 give 25%-33% of the credit for any problem left *completely blank* (for example, 1 point for a 3 point question). 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 problem—that is, any group of True/False items—is 0, and leaving an entire T/F problem (that is, all items) blank gets you 25%-33% of the credit for the entire problem.
- 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.*

| Name (4) | I (xx/10) | II (xx/36) | III (xx/16) | IV (xx/24) | V (xx/10) | Total (xx/100) |
|---|---|---|---|---|---|---|
| | | | | | | |

**Name (4 points):** Solutions **UT EID:**

## I Unix (10 points total)

**1. [4 points]** In the original Unix, as described in the papers that we read, hard links to directories were not allowed (besides the initial hard link that created the directory). In other words, each directory was allowed to have only one parent. This question asks about the motivation for this design decision.

**What could go wrong if directories could have multiple parents?**

If a directory could have multiple parents, then a directory *d* could create as a "child" a directory that was actually an ancestor of *d*, thereby creating a cycle in the file system hierarchy. Such cycles would introduce the need for garbage collection, which would add complexity.

**2. [6 points]** Recall that in Unix, the `fork()` system call creates a nearly-exact duplicate of the process that called `fork()`. Further recall that the `exec(filename, arg1, ...)` system call (more accurately, the variants like `execl()`, `execve()`, etc.) replaces the currently-running image with a new process image (specifically, `filename` is executed). The question we are asking here is: what is the power of having `fork()` and `exec()` be separate calls? The comparison is to a call like `spawn(filename, arg1, ...)` that simply creates a new running process by executing `filename`.

**Explain *briefly* the advantage of separating `fork()` and `exec()` versus having a single call like `spawn()`.**

The principal advantage that we discussed is that this separation permits arbitrary manipulation of the child's environment before the new code begins executing. The reason is that the same code block that called `fork()` also gets to execute *after* the `fork()`. That code can manipulate its own environment so that when it calls `exec()`, the new image has a different environment. With `spawn()`, the parent would need to manipulate the child's environment through suitable arguments to `spawn()`—which would limit flexibility since `spawn()` (or any pre-specified syscall) has a fixed interface. The examples that we saw in class were I/O redirection and pipes between processes.

**Name:** Solutions **UT EID:**

## II Virtual memory and virtualization (36 points total)

**3. [8 points]** In the paper "Virtual Memory Primitives for User Programs", Appel and Li quote another paper (Mike O'Dell, "Putting UNIX on very fast computers", Proc. Summer 1990 USENIX Conference):

> Modern Unix systems ... let user programs actively participate in memory management functions by allowing them to explicitly manipulate their memory mappings. This ... serves as the courier of an engraved invitation to Hell

**Explain *briefly* what the authors of this paper take this quotation to mean.**

See p.104 in the Appel-Li paper. Appel and Li interpret this quotation to mean that some authors have argued that it's a bad idea to allow user-level programs to manipulate their own view of memory (by, say, interposing on page faults). The reason it's a bad idea, according to Appel and Li's interpretation, is that, as a result of pipelining and out-of-order execution, the state of the program on a page fault may not correspond to program order. So if the page fault handler alters the CPU state (registers, say), the results may not be consistent with what the original code intended.

**Explain *briefly* why the authors say that the invitation to hell can be "returned to sender". (Hint: a proper answer to this sub-question is in two parts.)**

(1) Most of the algorithms that Appel and Li cover do not alter the CPU state; they simply unprotect a page. So out-of-order execution isn't really a problem. But that's not quite enough. (2) What's also needed is that, on a page fault, the hardware present a view to the page fault handler that is reasonable (even if that view includes out-of-order executions). But CPUs must already do so to support page fault handlers that page memory to and from the disk.

**4. [6 points]** State two reasons why paravirtualization (as practiced by Xen) performs so much better in execution time than full virtualization (as practiced by some of Xen's competitors).

**Be concise.**

(1) Batched update requests of the hypervisor, so amortize cost of trapping into hypervisor. (2) Fewer spurious page faults.

**5. [6 points]** Consider the x86 virtualization extensions that are measured in "A comparison of software and hardware techniques for x86 virtualization". Assume that the load on a given VMM is such that there is no need to page to the disk and likewise that the load on the guest VMs is such that there is no need to page to the virtual disk.

**Circle True or False for each item below, and note that the definitions below the items may be helpful:**

**True / False** These extensions provide absolutely no help to the authors in reducing *tracing faults*.

**True / False** These extensions provide absolutely no help to the authors in reducing *hidden page faults*.

Both are true. See section 4.2: "Since current virtualization hardware does not include explicit support for MMU virtualization, the hardware VMM also inherits the software VMM's implementation of the shadowing technique described in Section 2". Note that while the authors, at the end of the paper, discuss MMU virtualization, the question here mentioned the extensions that are actually *measured*.

**True / False** These extensions provide absolutely no help to the authors in reducing *true page faults*.

True. Given the assumption, the incidents of true page faults in a process are a function only of the program and its inputs. The virtualization technology, or even the presence of virtualization technology, is irrelevant.

**Relevant definitions**:

**Tracing fault:** A tracing fault occurs when (1) the guest OS attempts to modify the page tables that it sees while (2) the virtual address through which it access those page tables is, unbeknownst to that guest OS, not mapped in the shadow page tables that are currently pointed to from the real processor's `%cr3`.

**Hidden page fault:** A hidden page fault occurs when (1) the guest virtual machine attempts to gain access to a given virtual memory address while (2) that virtual address is, unbeknownst to that guest OS, not mapped in the shadow page tables that are currently pointed to from the real processor's `%cr3`.

**True page fault:** A true page fault occurs when a guest process attempts to gain access to a virtual memory address that is not mapped in the page tables pointed to from the *virtual* `%cr3`.

**6. [8 points]** This question also concerns "A comparison of software and hardware techniques for x86 virtualization".

**(a) Describe qualitatively (i.e., do not state the name of a benchmark) a type of workload for which the authors show that *hardware*-based x86 virtualization will perform better. (b) Explain *briefly* why the hardware-based techniques perform better on this workload.**

(a) Workloads with many system calls relative to I/O exits and page table modifications. (b) With hardware-based virtualization, the traps resulting from system calls go directly into the guest OS. In contrast, with software-based virtualization, these traps go first to the VMM, which then gives control to the guest OS; the extra transition is costly. Another answer is: (a) workloads with indirect control flow because (b) there is overhead from binary translation.

**(a) Describe qualitatively (i.e., do not state the name of a benchmark) a type of workload for which the authors show that *software*-based x86 virtualization will perform better. (b) Explain *briefly* why the software-based techniques perform better on this workload.**

(a) Workloads that cause many page table modifications. (b) With *both* software- and hardware-based techniques, page table modifications (as result from process creation, context switches, or normal growing, shrinking, and mapping of the process's address space) require the VMM to be involved (because of either tracing faults or hidden page faults). Assume for simplicity that we are using tracing faults, only. Then, without binary translation, whenever the guest OS modifies its own page tables, there is a costly trap into the VMM. With software-based techniques, the binary translator can avoid these traps by rewriting instructions that would modify page tables to simply apply the modification directly at translation time. See the paragraph labeled `ptemod` in section 6.3 of the paper.

**7. [8 points]** This question concerns "Memory Resource Management in VMware ESX Server". Recall that, in this paper, the VMM estimates the fraction of physical memory that each virtual machine (VM) is actually using, and recall that the VMM does so by statistical sampling, as explained in the third paragraph of section 5.3. The VMM uses this statistical sample as an input into which of the following decisions and calculations?

**Circle all that apply:**

**A** The choice of VM to reclaim a page from when using the ballooning technique.

**B** The choice of VM to reclaim a page from when using random paging.

**C** The choice of VM to inspect for possible content-based page matches.

**D** The choice of VMs that are subject to the "innocent until proven guilty" binary translation policy.

**E** How much to dock a virtual machine's shares, $S$, if the virtual machine hoards idle memory.

**F** The calculation of a virtual machine's adjusted shares-per-page ratio.

A., B., F. For C., content-based page matching happens regardless: all VMs are inspected for the possibility of such matching. D. is not covered in this paper. E. refers to a virtual machine's shares, which are set by the administrator, not by the VMM.

**Name:** Solutions **UT EID:**

## III Kernel structure (16 points total)

**8. [8 points]** This question concerns "Improving IPC by Kernel Design" and concentrates on Liedtke's Lazy Scheduling technique.

**Circle True or False for the two items below:**

**True / False** The gain from Lazy Scheduling, as a percentage of IPC time, depends heavily on the size of IPC messages.

True. See Table 1.

**True / False** Given Lazy Scheduling, the scheduler, when making a scheduling decision, is forced to scan all queues because threads could be on inappropriate queues.

False. Threads can be on inappropriate queues, but they are guaranteed to also be on the correct queue. The scheduler need only look at the current wake-up queue (or ready queue, depending).

Beginning at time $t_1$, the following sequence happens: a thread $A$ unblocks because a message is available, then processes the message, then replies to it, thereby blocking again for some time, then unblocks because another message is available, then processes that message, then replies to it, thereby blocking again. This last blocking begins at time $t_2$.

**Circle True or False for the two items below:**

**True / False** It is possible that, over the interval $[t_1, t_2]$, $A$ is never on the ready queue.

**True / False** It is possible that, over the interval $[t_1, t_2]$, $A$ is always on the ready queue.

Both are True. For the first T/F item, $A$ could have begun on a wakeup (or blocked) queue. When $A$ wakes up both times, control switches directly to $A$ with no insertion into the ready queue. For the second T/F item, $A$ could have begun on the ready queue. When it blocks, there is no requirement that $A$ be taken off the ready queue. The insight for both of these items is that the scheduler should refrain from moving a thread around because a thread that becomes newly runnable (resp., stops running) will probably go back to its original state, which is blocking (resp., running).

**9. [8 points]** Recall that in JOS, user-level environments have access to their own page tables, through the virtual addresses $[\texttt{UVPT}, \texttt{UVPT} + 4\text{MB})$. However, this access is read-only.

**If an environment were permitted to write to its page tables, how could that environment harm the system or other environments?**

An environment could create arbitrary (VA,PA) mappings (that is, to arbitrary physical addresses), which would allow an environment to read and write the memory of another environment or of the kernel.

In Aegis, as described in "Exokernel: An Operating System Architecture for Application-Level Resource Management", an application (which is analogous to a JOS environment) has both read *and* write access to its page tables.

**Why is it safe for applications to write to their page tables in Aegis whereas it is not safe in JOS?**

The fundamental reason is that Aegis ran on the MIPS, which has a software-managed TLB (and the hardware is unaware of the page tables), whereas JOS runs on the x86, which has a hardware-managed TLB and hardware-visible page tables. On JOS, a spurious entry in the hardware-visible page tables would compromise isolation. On Aegis, a spurious entry in the page tables has no effect on safety. What enforces safety is that Aegis regulates the TLB entries by ensuring that, when an application calls `TLBWr(VA,PA)`, the page given by `PA` has actually been allocated to the application. Note that, were Aegis to run on the x86, it would not be able to grant applications write access to their page tables. Conversely, were JOS to run on the MIPS, applications could safely get writable access to their page tables. Discussions of "secure capabilities" are irrelevant: the term is basically fancy terminology for "check that this environment is allowed to set up that mapping". JOS has an analogous thing: in `sys_page_map`, JOS checks that the calling environment has the privilege to map the given page (by checking whether there is a needed parent-child or self-self relationship between the two environments). This check is not dissimilar to validating the parameters to `TLBWr(VA,PA)`.

## IV Concurrency and scheduling (24 points total)

**10. [8 points]** This question concerns monitors and condition variables, as described in "Experiences with Processes and Monitors in Mesa" and refined by "Programming with threads", by Mike Dahlin. If you believe that the Mesa paper and Mike Dahlin's document conflict, follow Mike Dahlin's document; deviating from the standards there is considered incorrect. Recall that programmers who use condition variables are given a set of primitives:

```
Cond::wait();
Cond::signal(); // called "notify()" in Mesa
Cond::broadcast();
```

Assume that `Cond::wait()` does not take a timeout argument and that there is no facility for aborting a process or thread that is `wait()`ing. Define the *programmer* to be the user of condition variables, threads, processes, locks, etc. Define the *implementation* to be the code that implements condition variables, threads, processes, locks, etc.

**Circle True or False for each item below:**

**True / False** The Mesa scheduler is preemptive.

True.

**True / False** There are cases when it is correct for the programmer to `signal()` a condition variable without holding the monitor lock.

Credit for True, False, or blank. The intended answer was False because MikeD's document says, "Always hold lock when operating on a condition variable". However, some people thought that this question was asking about the naked notify, as described in Mesa. It's true that a naked notify is a `signal()` without holding a lock, but this notify is done by the hardware, not the software (also, we said that if MikeD's document and Mesa conflict, go with MikeD's document). Still, we did not define "programmer" sufficiently to exclude the hardware, so we gave credit for all answers.

**True / False** Assume that several threads are inside `wait()`. If no `signal()` or `broadcast()` has taken place, but the implementation returns one of the threads from the `wait()` call, that is a bug.

False. `wait()` must be held inside a while() loop. The while condition protects safety, giving the implementer (and the programmer!) the freedom to issue spurious notifications. This point was also made in class.

**True / False** Assume that several threads are inside `wait()`. If a `signal()` takes place, but the implementation ignores it (by not placing any thread in the runnable state), that is a bug.

Credit for True, False, or blank. The intended answer was True: if signals were permitted to be dropped, then there would be no way to ensure liveness when using `signal()`. Although Mesa describes `signal()` as a hint, the hint is in one direction: under the assumption that there are no timeout calls and that `abort()` is unimplemented, it's okay to issue too many `signal()`s but not too few. The reason that we gave credit for other answers is that the problem does not specify that the `wait()` and the `signal()` were being done on the same condition variable.

**11. [8 points]** This question concerns "Borrowed-Virtual-Time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler". Assume that the operating system implements a single-level BVT scheduler and that each thread has a single warp value set at thread creation time. Further assume that the operating system provides monitors. *There are four True/False items in this problem.*

**Circle True or False for the three items below, and note that the definitions at the end of the question may be helpful:**

**True / False** Under BVT, starvation is possible.

**True / False** Under BVT, priority inversion is possible.

**True / False** Under BVT, deadlock is possible.

The first is False, ensured by the weights and the scheduling discipline: in the long run, every thread gets to run. The second two are True. The priority inversion example given in Mesa is possible in BVT, given our definition of priority and given that each thread gets only one warp value. And deadlock results from programming bugs that are independent of BVT.

**Now consider the following scenario:**

At time $t_a$ microseconds ($\mu$s), a thread $T_1$ is given the processor. At time $t_b = t_a + 1\mu$s, a thread $T_2$ becomes runnable. The context switch allowance, $C$, is 20 milliseconds, and *mcu*, the minimum charging unit, is 5 milliseconds. Both threads have the same weight.

**Circle True or False for the item below:**

**True / False** If, at time $t_b$, the effective virtual time of $T_2$ is smaller than the effective virtual time of $T_1$, then the scheduler stops running thread $T_1$ at $t_b$.

True. The context switch allowance is not considered when a thread has just woken up. The point to the effective virtual time is to allow low latency threads to run with low latency. Thus, when a thread wakes up, if it is has low effective virtual time, it runs immediately. (As an aside, note that in the example given, $T_2$ doesn't necessarily get to run: there might be another thread, $T_3$, with a lower effective virtual time than that of $T_2$.)

## Definitions

**Starvation:** One thread permanently preventing another from executing.

**Priority inversion:** A medium priority thread (where priority is defined below) prevents a high priority thread from executing, when neither is waiting for I/O. This can happen if the high priority thread is waiting for a resource held by a low priority thread. The Mesa paper gives an example of priority inversion.

**Priority:** In BVT, what it means for a thread $T_i$ to have higher priority than a thread $T_j$ is that if both $T_i$ and $T_j$ are runnable, and if neither has run in some time, then $T_i$ has lower dispatch latency, that is, the scheduler will choose $T_i$ in preference to $T_j$. This preference is expressed by giving $T_i$ a higher warp value than $T_j$.

**Name:** Solutions **UT EID:**

**12. [8 points]** This problem concerns Determinator, as described in "Efficient System-Enforced Deterministic Parallelism". Assume that Determinator is running on a single processor machine.

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

If two or more Determinator *spaces* (as defined in section 3) are runnable (that is, not blocked inside `Put`, `Get`, or `Ret`), some module in Determinator must assign a space to the processor for some length of time. Call this module the *space scheduler*.

**Circle True or False for both of the items below:**

**True / False** When choosing a space to run next, Determinator's space scheduler is permitted to choose randomly. (By *random*, we mean that the choice is based on a source of external randomness, such as nuclear decay as detected by a Geiger counter.)

**True / False** When choosing the maximum length of time that the chosen space will be allowed to run, Determinator's space scheduler is permitted to choose randomly. (By *maximum length of time*, we mean the length of time after which the space will be preempted, if it has not yielded or blocked.)

Both are true. The determinism is enforced by `Put`, `Get`, `Ret` and the programming model. Arbitrary interleavings of runnable threads will still produce the same output in Determinator, thereby not compromising its determinism. Geiger counters aside, Determinator's scheduling is presumably driven in part by an external timer source, so we would not expect its scheduling decisions to be deterministic (except to the extent that today's computers are fundamentally deterministic machines).

Now consider *processes* (section 4.1), which are built on top of Determinator's spaces. Each process runs on a single space. Assume that, for synchronization, the processes use only deterministic synchronization primitives (for instance, `fork()`, a deterministic version of `wait()`, and barriers—but not mutexes).

**Circle True or False below:**

**True / False** The interleaving of processes on the processor is not permitted to depend on an external source of randomness. (By *interleaving*, we mean a conceptual schedule: one can imagine listing which process gets the processor and for how many instructions, then which process next gets the processor and for how many instructions, etc. This conceptual list is the interleaving, or the schedule.)

False. The programming model is deterministic and remains so regardless of the scheduling order. This is False for the same reason that the first two are True.

Now imagine that a single process $P$ uses Determinator's runtime (which itself uses spaces) to instantiate several *threads* (section 4.5). Assume that the threads use nondeterministic synchronization primitives (mutexes, condition variables, etc.).

**Circle True or False below:**

**True / False** The interleaving of $P$'s threads is nondeterministic. (By *nondeterministic*, we mean that if $P$ is run multiple times with the same input, the interleavings might be different.)

Credit for True, False, or blank. In Determinator, the goal of the user-level scheduler is to produce what it *thinks* is a deterministic interleaving of instructions, per Section 4.5. So if your interpretation concerned how the user-level scheduler schedules processes on threads, the answer was False. However, the answer to the question as literally posed was True. The reason is that each user-level thread is assigned to a space, so the non-deterministic scheduling of spaces means that the actual interleaving of each thread's instructions is actually non-deterministic (for the same reason that the first three items above had the answers that they did).

## V JOS and feedback (10 points total)

**13. [8 points]** Recall that the JOS kernel runs non-preemptively. That is, when kernel-level code is executing, interrupts are *never* enabled. This guarantee ensures that the kernel will never itself be interrupted by, say, a device receiving data. On the other hand, when user-level code is executing, interrupts are *always* enabled. This guarantee ensures that the kernel can regain control from user-level processes (as is necessary on timer interrupts, device I/O, illegal instructions from a user-level process, page faults in a user-level process, etc.)

**What code did you write and/or what data structures did you set up to ensure that interrupts are *never* enabled in kernel mode? Be specific and concise.**

See lab 3.

**What code did you write and/or what data structures did you set up to ensure that interrupts are *always* enabled in user mode? Be specific and concise.**

You took care of this in lab 4.

**14. [2 points]** This is to gather feedback. For both of the questions below, any answer will receive full credit. *A blank answer will earn zero points.*

**Please list the two papers or classes in this course that you have most enjoyed:**

There is no right answer.

**Please list the two papers or classes in this course that you have least enjoyed:**

There is no right answer.

End of Midterm