CSE421 Final Exam Solutions

—SOLUTION SET—

07 May 2012

This final exam consists of four types of questions:

1. **Ten multiple choice** questions worth one point each. These are drawn directly from second-half lecture slides and intended to be easy.

2. **Six short answer** questions worth five points each. You can answer as many as you want, but we will give you credit for your best four answers for a total of up to 20 points. You should be able to answer the short answer questions in four or five sentences. These are also drawn from second-half material exclusively.

3. **Two medium answer** questions worth 20 points each, also drawn from second-half material. **Please answer *one*, and only one, medium answer question.** If you answer both, we will only grade one. Your answer to the medium answer should span a page or two.

4. **Three long answer** questions worth 25 points each, integrating material from the entire semester. **Please answer *two*, and only two, long answer questions.** If you answer more, we will only grade two. Your answer to the long answer question should span several pages.

Please answer each question as **clearly** and **succinctly** as possible. Feel free to draw pictures or diagrams if they help you to do so. The point value assigned to each question is intended to suggest how to allocate your time. So you should work on a five point question for roughly five minutes.

No aids of any kind are permitted.

Multiple Choice

1. (10 points) Answer all **ten** of the following questions. Each is worth **one** point.

   (a) Which of the following did not happen during lecture this semester?
   ◯ Some students refused to stand-up. √ **Geoff dropped the f-bomb.** ◯ Chuchu wandered out of the lecture hall. ◯ We checked the temperature in Berkeley.

   (b) A head crash occurs when
   ◯ two disk heads collide while scanning across the disk. √ **a violent movement of a spinning disk causes the heads to make contact with the platter while spinning.** ◯ Geoff's head hits his desk after writing this exam. ◯ a disk head crashes into the spindle.

   (c) All of the following are on-disk filesystem data structures *except*
   ◯ inodes. ◯ data blocks. ◯ the superblock. √ **vnodes.**

   (d) Kirk McKusick is
   √ **the developer of the Berkeley Fast File System (FFS).** ◯ Captain Kirk. ◯ the developer of the Log-Structured Filesystem (LFS). ◯ dedicated to using technology to monitor his extensive beer collection.

   (e) Cylinder groups are an idea for improving spinning disk performance introduced by
   √ **FFS.** ◯ LFS. ◯ ZFS. ◯ ReiserFS.

   (f) It is possible to prove the correctness of a microkernel.
   √ **True.** ◯ False.

   (g) All of the following describe ways to structure an operating system kernel *except*
   ◯ multikernel. √ **zetakernel.** ◯ exokernel. ◯ monolithic kernel.

   (h) When your performance data has outliers, you should
   ◯ assume they are experimental error and ignore them. ◯ delete them and never speak of them again. √ **love and try to understand them.** ◯ have never discovered them by exclusively using summary statistics.

   (i) Which of the following is *not* a useful approach to improving system performance?
   ◯ Carefully choosing an appropriate benchmark. √ **Improving the parts of your code that you just know are slow.** ◯ Analyzing data from experiments to identify bottlenecks. ◯ Developing a new simulator to improve reproducibility.

   (j) Which of the following is not one of the requirements of implementing a hardware virtual machine?
   ◯ Fidelity. ◯ Performance. √ **Atomicity.** ◯ Safety.

Short Answer

Choose **four of the following six** questions to answer. You may choose to answer additional questions, in which case you will receive credit for your best four answers.

2. (5 points) Using appropriate terminology, explain the process of moving from one location to another location on a spinning disk. Identify each major source of latency.

Solution:

The full list of steps presented in class is as follows:

1. **Issue the command.** The operating system has to tell the device what to do, the command has to cross the device interconnect (IDE, SATA, etc.), and the drive has to select which head to use.
2. **Seek time.** The drive has to move the heads to the appropriate track.
3. **Settle time.** The heads have to stabilize on the (very narrow) track.
4. **Rotation time.** The platters have to rotate to the position where the data is located.
5. **Transfer time.** The data has to be read and transmitted back across the interconnect into system memory.

Given that the question asked about moving from one location to another and did not mention the operating system or performing an action that would require transferring data, answers that only included **seek time**, **settle time** and **rotation time** are acceptable, and potentially even more correct.

Sources of latency are seek time (major), settle time (minor), rotation time (minor). An answer that only identified seek time as a *major* source of latency is acceptable.

3. (5 points) Describe how filesystem journaling works:

   - What is written to the journal?
   - How is the journal used after a crash to quickly return the filesystem to a consistent state?

Solution:

A filesystem journal should record *all changes* to the filesystem state and data structures. The only exception to this inclusivity is data blocks, which can be written to the journal or may not be. (In the latter case the filesystem may lose data after a crash but should still maintain consistency.)

Examples of changes always recorded in the journal would be data block and inode allocation, changes to inodes, changes to internal data structures such as inode or data block allocation bitmaps, and changes to the superblock.

Checkpoints are written to the journal periodically and indicate that all changes recorded previously have been written to disk. When a crash occurs, the filesystem uses the journal to recover a consistent state. It does this by scanning the journal starting at the latest checkpoint and looking for uncommitted operations: those that are written to the journal but not to disk. Once all uncommitted operations have been flushed to the disk, the filesystem should be in a consistent state and can resume operation.

4. (5 points) Identify the difference between a *write-through* and *write-back* block-level filesystem buffer cache. Which is better for safety? Which is better for performance?

Solution:

A *write-through* cache does not buffer writes. Instead, they are passed through to the disk immediately. Write-through caches improve read performance—reads still hit the cache—but do not improve write performance. However, they are safer as all write operations are on disk as soon as possible.

A *write-back* cache does buffer writes. Modifications are not written to disk until blocks are evicted. (Write-back caches may also write data in the background or ensure that blocks are flushed after some configurable delay.) Write-back caches provide faster write performance but are less safe, as modifications may not be lost in the cache if a failure occurs.

5. (5 points) List several of the impacts that micro-kernel design and development has had on modern monolithic operating systems.

Solution:

1. Increased attention to what really belongs in the kernel and what doesn't. This doesn't always work out in practice in monolithic kernel, but it's useful to think about.
2. Work on very fast inter-process communication (IPC). Since microkernel's live and die by the speed of process-to-process transitions and communications, this is an area where they focused their efforts.
3. Attention to interfaces between kernel components. Monolithic kernel can benefit from clean, well-designed interfaces between core components even if everything runs in the same address space.

6. (5 points) Define Amdahl's Law and describe how it guides the process of performance improvement.

Solution:

We discussed two different formulations of Amdahl's Law:

> The impact of any effort to improve system performance is constrained by the performance of the parts of the system **not targeted** by the improvement.

—or—

> Ignore the thing that **looks** the worst and fix the thing that is **doing the most damage.**

We also had our corollary to Amdahl's Law:

> The more you improve one part of a system the less likely it is that you are still working on the right problem!

Amdahl's Law informs performance improvement in a variety of ways. It says that if you aren't working on the right problem, you aren't going to accomplish much, meaning that it's essentially to figure out *what* the right problem is. It also implies that once you've made an improvement, you need to reassess because another part of the system may now be your bottleneck.

7. (5 points) Assuming full hardware virtualization, describe how an application-generated TLB fault is handled by the guest operating system. Assume that this virtual machine architecture supports a software-managed TLB, and that the page causing the fault is already resident in the virtual machine's memory.

Solution:

1. The original TLB fault happens in the virtual machine (VM). Control immediately jumps to the *host* operating system.
2. The host notices that the fault has occurred in the VM and delegates handling of the fault to the virtual machine monitor (VMM).
3. The VMM notes that this as a TLB miss which must be handled by the *guest* OS, and vectors control to the guest operating system's TLB fault handling code.
4. The guest OS will begin executing believing that the original TLB fault vectored control to it immediately. It will look up the correct translation for the process running in the VM that generated the fault and attempt load it into the TLB.
5. The attempt to load the TLB by the guest OS will generate another fault because it is not running in privileged mode. This second fault will again be vectored by the host OS to the VMM.
6. The VMM will inspect the TLB write and deem it to be safe if the physical address that the translation points to is physical memory allocated to the VM. In this case, the TLB modification will complete, and the process running inside the VM will be restarted.

Medium Answer

Choose **one of the following two** questions to answer. **Please do not answer both questions.** If you do, we will only read one.

Complete this question on the scratch paper attached to the back of the exam. Clearly label your answer.

8. (20 points) Label your medium answer as **Question 8**.

Virtual Machine `fork()`

You know all there is to know about `fork()`, of course, but your boss at your new software development job just discovered `fork()` and thinks it is just the coolest thing. Unfortunately he also just learned about virtual machines (the second coolest thing), and has hatched a plan to combine the two. Or, to put it accurately, his plan to combine the two is to ask *you* to implement an analog to `fork()` but one that forks *virtual machines* instead of processes.

First, describe virtual machine `fork()` at a high level. Who would implement virtual machine `fork()`? How would they do it? Identify several significant differences between virtual machine `fork()` and process `fork()` arising from the differences between processes and virtual machines. (You may want to think about the rest of the process-related system call interface—`exec()`, `exit()`, and `wait()`—and consider there are virtual machine analogs to these process operations as well.)

Second, explain why this may or may not be a good idea. You may want to consider use cases and overheads.

Solution:

At a high level, virtual machine `fork()` is similar to process `fork()`: the virtual machine the called `fork()` is stopped, a new virtual machine is created that is a copy of the machine that called `fork()`, and both virtual machines are restarted. We create the copy in in a similar way to the way that we copy address spaces in process `fork()`, except that the mappings that need to be copied are guest OS physical → host OS physical rather than process virtual → physical.

The differences begin to emerge, however, after we consider *who* and *how* virtual machine `fork()` would be implemented. Because this is an operation that acts on the *virtual machine*, it would have to be added as a *hardware feature* and implemented by the Virtual Machine Monitor (VMM). Thus, users of the virtual machine would have to be made aware of this feature, and this awareness itself would pierce the virtual machine.

In addition, since processes are themselves an abstraction, process `fork()` can "copy" the process abstraction. But physical machines—which the virtual machine is trying

to copy—are *not* abstractions, and cannot be instantly copied, and so virtual machine `fork()` *must* "pierce" the virtual machine. Thus, on some level virtual machine `fork()` is fundamentally incompatible with full virtualization, which we discussed in class. Paravirtualization—which requires changes to the operating system code that make it aware of the virtual machine monitor hypervisor—is a better fit for this abstraction. (You are not be expected to have mentioned paravirtualiation since we did not cover it in lecture.)

Other differences fall out when we examine the rest of the process-related system calls. Given the idea we presented of a virtual machine, many of the features of the process system calls designed to enable communication between processes don't really make sense, since a fully-virtualized virtual machine is not suppose to provide its guests any idea that there are other virtual machines using the same physical machine. (This is the definition of piercing the VM.) So support `wait()/exit()` semantics, for example, would require fundamentally altering what VMs are expected to know about their world.

As far as evaluating this idea, you need to consider what it costs and what it would be used for. Copying an entire virtual machine is potentially very expensive, since all of the operating system code must be copied along with the state of every running application. On the other hand, it is exactly that copying of the *entire state* of the virtual machine that makes virtual machine `fork()` potentially useful. So, for example, an application distributed across multiple machines could create (and destroy) new virtual machines as demand fluctuated. The new virtual machines would benefit from not having to boot or restart many applications, which could take significant amounts of time.

Besides the cost, another limitation of virtual machine `fork()` is simply the lack of good inter-virtual-machine communication mechanisms analogous to IPC for multiple processes. Many times parent-child pairs want to communicate in order to do useful work, with the parent calling process `fork()` and then telling the child what it should do, for example. There will also likely be complications concerning the state of the machine that cannot be copied—hardware addresses like the NIC MAC addresses, the IP address that the machine has established—and other things that might depend on this state, such as open TCP connections, which depend on the IP address.

Log-structured File Systems on Flash Drives

Flash drives have very different performance characteristics than spinning disks. One important difference is that, because Flash drives contain no moving parts, latencies for access to different locations on disk are roughly constant.

Flash drives do, however, have drawbacks. One concerns how writes are performed. In order to write a byte to a block on a Flash drive you must first erase the entire *erase unit* that contains it, which are usually quite large. Given an erase unit of 16 Kb, the process of modifying a single byte becomes:

- Read the entire 16 kB erase unit into memory in order to preserve the unmodified contents across the erase.
- Change the byte we want to change in our cached copy in memory.
- Erase the erase unit.
- Write the entire modified erase unit back to the Flash drive.

The other important characteristic of Flash drives is that the individual erase units can only be erased a fixed number of times, making it important to try and even out the usage of different portions of the disk.

It seems like it's time for log-structured filesystems to make a comeback, and maybe Flash drives are the place! Let's think this through. Begin by explaining why this may seem like a questionable choice by identifying the key benefit of log-structured filesystems on spinning disks that is lost on Flash. But continue by discussing how log-structured filesystems might help address the two challenges of Flash devices outlined above.

Solution:

Recall that the big benefit of log-structured filesystems was that by performing all writes to the log they reduced the seek time for writes, which tend to dominate disk traffic if we assume that many or all reads are absorbed by large buffer caches. (Log-structured file systems were born into the world of growing main memory sizes but stubbornly-slow disks.) On Flash drives this argument makes no sense, since there are no moving parts and no seek times, so the advantage of performing all writes to the same location on the disk is lost.

However, what is not lost is the other advantage of performing all writes to the same location, which is that the filesystem can *aggregate* the writes in the cache and perform a bunch of writes all at once to the same location. On Flash drives, this allows the filesystem to hold all writes to the same erase unit until the entire erase unit is full, at which point it can be erased and rewritten as a single chunk. (This is what has to happen anyway.)

To use an example, suppose we have 16 writes to perform that together will fill up an erase unit. On a traditional filesystem on Flash, those 16 writes will—barring some

other clever technique—land in 16 separate erase units, causing each erase unit to have to be read into memory, erased, and then rewritten with on average $\frac{1}{16}$-th of the erase unit modified. Using a log-structured file system, assuming a clean erase unit all writes will land in the same erase unit, causing a single erase and rewrite cycle.

Logs can also address our wear-leveling challenge as well. Ignoring log cleaning temporarily, which I know is a stretch, you would expect the log to simply rotate across the disk at the speed of ongoing write traffic, evening out writes to the highest degree possible. And there are probably clever ways to clean the log as well that also evenly-distribute write traffic.

Both advantages above share a similar structure. Log structured file systems improve write locality. On spinning disks this helps with seek time. On Flash drives, this means the disk can perform *fewer* writes and also can exercise *more control* over where those writes go, leading to longer happier Flash disk lifetimes.

Long Answer

Choose **two of the following three** questions to answer. **Please do not answer all three questions.** If you do, we will only read the first two.

Complete this question on the scratch paper attached to the back of the exam. Clearly label your answer.

9. (25 points) Label your first long answer as **Question 9**.

10. (25 points) Label your second long answer as **Question 10**.

Application Prefetching

Recall that on systems that use demand paging, page contents are not loaded into memory when the process address space is initialized during `exec()`. Over the lifetime of a process, this procrastination-based technique has significant benefits. In particular, pages containing code that is never executed by the application are never paged in to or swapped out of memory.

One potential downside, however, of demand paging is that when an application first begins executing it generates a large number of page faults to bring the code and libraries it does use in from disk. Application prefetching is a technique supported by several modern operating systems to reduce application startup times on systems with big, slow, spinning disks by *using the past to predict the future*.

First, assuming that the code pages needed by the application as it loads are scattered all over the disk, describe the potentially deleterious consequences of performing these I/O requests in the order the process generates them on a large, spinning disk.

Second, explain why operating systems can accurate predict an application's startup I/O requests. To answer this question, you might consider what is fundamentally different about the first few seconds of an application's execution from other time periods and how that would make its I/O patterns more predictable.

Finally, use these two observations to design application prefetching.

1. Explain what information you need to know about an application to prefetch effectively and how to collect that information.
2. Describe what happens when a prefetched application begins running and how it differs from a non-prefetched application.
3. Present an argument as to why your prefetcher improves application startup performance.

Solution: Application Prefetching

First, the potentially "deleterious" (a.k.a, bad) consequences of on-demand paging during program startup is that, assuming the code pages that it needs to initialize are distributed randomly across the disk, the disk head will be sent on a random walk across the platter, incurring high overheads for each successive seek.

Second, the key insight that enables application startup prefetching is that, while demand paging patterns during interactive program execution are inherently unpredictable—they depend on what the user is doing—the first few seconds of interactive program execution are highly-predictable. During this time the program is loading libraries, performing initialization routines, and generally doing ignoring the user while it prepares itself to run. For some applications—think of Adobe PhotoShop—this phase can continue for quite some time.

Finally, we design an application prefetcher as follows. What we need to know is all of the disk I/O's that a process will generate during the first phase of predictable execution. We can collect this information by profiling the application during startup time. This profiling phase can also be used to identify *how long* the predictable initialization period is by comparing successive executions of the same binary program. We store this information in a *prefetch file* for each application.

When a prefetched application begins running, rather than allowing it to demand-fault each page in, we use the *prefetch file* to prefetch all of the code and library pages it will need during its initialization period. So, in contrast to a non-prefetched application, a prefetched application will generate page faults during initialization. Instead, it should find these pages in memory. Of course, this is transparent to the process (modulo timing), since TLB faults will still occur but simply not generate page faults since the pages all already resident in memory.

The argument that this improves performance is fairly simple to construct and relies on an understanding of disk head scheduling. Presented with single, randomly-located I/O requests, the heads perform their random walk and performance suffers. Presented with a large number of I/O requests, the head scheduler can sort them and do a *single pass* across the platter, collecting all of the required data on the way. This is much faster, and the Windows prefetcher was able to reduce startup times of large applications significantly. (The same technique was applied to Windows boot as well, with similar results, bringing us ever-closer to the system that takes zero time to reboot.)

Asynchronous System Calls

Throughout the semester we have considered system calls as being *synchronous*, or blocking: when a process performs a system call, it is blocked until the call completes. However, modern operating systems also support *asynchronous*, or non-blocking, system calls. These allow a process to request the operating system perform some action on its behalf while not requiring the process wait for the action to take place.

First, considering the system calls we have discussed throughout the semester, describe several cases in which asynchronous system calls would be useful and how. Conversely, describe several synchronous system calls that lack a meaningful asynchronous analog.

Second, describe any changes to the operating system interface that might be necessary to support certain asynchronous system calls. Discuss any additional application programming challenges that non-blocking system calls may introduce.

Third, briefly explain how a process that can fork multiple threads can emulate asynchronous system calls without true non-blocking support from the operating system. Describe the overheads to this approach that might make native asynchronous system calls preferable.

Finally, describe the kernel changes necessary to support asynchronous system calls. Walk through the steps required to complete a non-blocking call, describing what happens at both the process and kernel level.

Solution: Asynchronous System Calls

1. Asynchronous system calls are particularly useful when performing I/O. A process can issue `read()` and `write()` calls without waiting for those calls to complete, allowing it to go about other business while waiting. (This can be particularly useful when implementing so-called *event-based* programming frameworks, which provide an alternative to the multi-threading programming model most of us are more familiar with.)

   Another case of a system call that is not I/O-related that has a meaningful asynchronous analog is `waitpid()`. Here it is helpful to allow the parent process to return immediately if the child process has not exited. We refer to a process repeatedly checking on a result, as it would be calling an asynchronous non-blocking `waitpid()`, as *polling*.

   Several system calls, however, really don't make much sense asynchronously. What does it mean to do an asynchronous `exec()` or `fork()`? In the former case, the process just returns and can run for a bit longer before it is replaced by a new image. This doesn't seem useful. With `fork()` you have the problem that the parent wants to create a copy at a well-defined moment, and allowing it to return and continue running would complicate that process. (You could do this though, if you were careful, but again the use case doesn't seem obvious.)
2. Let's focus on the I/O related asynchronous calls, particularly `read()` and `write()`. What would we need to add in order to allow the process to return immediately? The first thing is that we need to provide a way for the process to determine *when the call has completed*. One way is to provide some additional system call that a process can use to poll for a pending I/O request; another way is for the operating system to send the process a signal when the requested I/O completes.

   The reason that this additional signaling mechanism is required is that the process must not modify or interpret the `write()` or `read()` buffer until the non-blocking call completes. This requirement also creates a new concurrent programming challenge which the application designer must deal with. With a blocking system call, there is no way for the process to see a `read()` in an incomplete state; with a non-blocking call it can, and care must be taken to ensure that it does not assuming the correctness of the program depends on it. (It probably does; imagine a web-server that served pages that had not been completely read from disk.)
3. If a process can create threads visible to the kernel it can emulate asynchronous calls. When a thread wants to perform a blocking system call, it forks a new thread specifically for this task. That second thread blocks across the system call while the original thread continues to run. When the call completes, the thread that was forked to perform the call exits.

The problem with this approach is that there can be a potentially high overhead to forking threads that are visible to the operating system kernel. Note that user-only threads are not appropriate for this purpose. It would be great if they were, because the overhead of forking a user thread is significantly lower, but because they are not visible to the operating system a user thread that blocks will stop the entire process and negate the objective we were trying to achieve.

4. When a process performs a non-blocking call, assuming the arguments are correct and the `read()` or `write()` can be initiated, the kernel can fork a separate kernel thread to complete the request, allowing the calling thread to return to user mode. The delegated kernel thread is queued to wait for the I/O to complete. When it awakens, it must take any actions necessary to notify the process that the asynchronous I/O has completed: marking it as done in a data structure that the process can poll or sending the process the appropriate signal. Kernel threads used for this task may be created when non-blocking calls begin and destroyed when they are complete, or they may be part of a dedicated kernel *thread pool* and recycled across subsequent calls.

Operating System Transactions

Transactions are a database concept and, for the purposes of this problem, can be defined as a set of actions that must be **a**tomic, **c**onsistent, **i**solated and **d**urable. (Together these requirements are known as **ACID** semantics.) We will focus on two of the ACID properties: atomicity and isolation. Atomicity requires that transactions either either *all succeed* or *all fail*. Isolation requires that the changes made by a single transaction not be visible in an intermediate state.

Traditional operating systems interfaces do not provide support for transactions spanning multiple system calls. TxOS, a new research operating system developed at the University of Texas, asserts that adding transaction support will help solve a number of common operating system problems. One common problem is helping keep multiple related files synchronized—adding a user to a UNIX system requires changes to `/etc/passwd`, `/etc/shadow` and `/etc/group`, and certain utilities may fail if they read these files during an update and find inconsistent state: a user without a password during user creation, or a group containing a user that does not exist during user deletion.

First, describe how one of the two cases above can occur. List the system calls in pseudocode for the two processes involved and how they would interleave in time leading to one process viewing inconsistent state. You can assume that both adding and removing a user requires modifications to all three files mentioned.

Second, describe a solution that provides support for transactions consisting of multiple modifications (via `write()`) to one or potentially multiple files. (This is much easier than providing transaction support for the entire filesystem or operating system interface!) Assume that we have added two new system calls `sys_xbegin()` and `sys_xend()` delineating the beginning and end of a single transaction. Your solution should meet the following design requirements ensuring *atomicity* and `isolation`:

1. Any writes to files made by a process after calling `sys_xbegin()` should not be visible until that process calls `sys_xend()`.
2. Any files modified by the transaction can only be modified if they are in the same state when the process calls `sys_xend()` as they were when it called `sys_xbegin()`. You must ensure this by either preventing them from being changed, or by aborting the transaction if you detect that changes have occurred.
3. To achieve good performance you should attempt to keep any locking as fine-grained as possible.
4. It is acceptable, and a property of many good solutions, to fail transactions, but there should be reasonable conditions under which a reasonable transaction can succeed. (Definitions of reasonableness are yours to provide.)

Finally, explain how operating system transactions might be useful when updating system software.

Solution: Operating System Transactions

1. **A user without a password during user creation.** The process creating the user writes the entry to `/etc/passwd`. Another process reads `/etc/passwd` and finds the entry for the user but does not find an entry in `/etc/shadow`. Finally, the process creating the user writes an entry to `/etc/shadow`.
2. **A group containing a user that does not exist during user deletion.** The process deleting the user removes the user entry from `/etc/passwd` (and `/etc/shadow`). Another process reads `/etc/group` and finds a group containing the user but does not find the user in `/etc/passwd` or `/etc/shadow`. Finally, the process deleting the user removes the user from all groups in `/etc/group`.

There may be multiple ways to design solutions for this question. Here is the one we had in mind.

When a process calls `sys_xbegin()`, the operating system (1) notes the time that the transaction began and (2) marks that the process is performing a write transaction. Saving the transaction start time is critical later to determine whether files have been modified *after* the transaction started which should cause the transaction to abort.

When a transaction is in process and a process calls `write()` on a file for the first time, the operating system first checks whether any other writes have occurred to that file (by other processes) since the transaction began. If the answer is yes, we must abort the transaction as we have violated our **atomicity** property.

If the answer is no, we must first flush any dirty buffer cache blocks for this file and then lock the file for the process in the transaction, preventing *future* writes and reads to the file until the transaction completes. In order to abort cleanly, we will also buffer all writes during the transaction in the buffer cache. In case of abort, we will simply drop all buffers associated with the transaction, returning all files to their on-disk state, or the state that they were in when the transaction began, ensuring atomicity.

When files are locked by a process performing a transaction, other writes and reads from other processes will be delayed until the transaction completes. On some level this seems to enable out-of-order writes, but in reality we are creating the illusion to other processes that all of the writes occurring as part of the transaction occur *at the same time* and at the time that the call to `sys_xbegin()` was made. We could even adjust the file modification times appropriately. Reads must be stopped to prevent other processes from seeing modifications to the files that have not been committed yet. (An alternate—and more clever—solution would allow the reads to complete but provide data consistent with the start of the transaction.)

When the process performing the transaction calls `sys_xend()`, we (1) flush all dirty buffers written as part of the transaction to disk and (2) drop all file locks that the process has acquired.

Returning to our requirements and suggestions, we note that our solution meets them. We meet the modification requirement through two mechanisms: aborting on the first write if the file has been changed after the transaction, and locking after the first write. Part of the difficulty here that causes us to have to sometimes abort is that we *do not know* what files the process will use during the transaction when the transaction starts. If we did, we could grab all the locks up front. Since we don't, we grab the lock as early as possible.

Second, we keep our locking as fine-grained as possible by locking on the file level. Grabbing one big filesystem wide lock is not a solution that is going to earn a lot of points for this question, although it is a potential solution and so will earn some points. You should be able to do better than that!

Note that we will fail transactions in certain cases, with the likelihood of failure rising as the number of files, number of modifications and "length" (in time) of the transaction rises. The more time that passes between the initial `sys_xbegin()` and the transaction-initiating process's first access to a file, the higher the chance that another modification has occurred during that time. So our expectation is that transactions involving small numbers of changes to a few files will succeed often under normal load.

Finally, operating system transactions are useful when installing system software because an entire set of related changes can be performed and, if one package fails, the system can be rolled back to its pre-installation state. Without transaction, what usually happens is that bits and pieces of the failed installation are left everywhere, potentially complicating future software package installations. Many systems support other mechanisms to try and address this specific problem—like Windows checkpoints—without implementing full operating system transaction support.