CSE421 Final Exam Solutions

—SOLUTION SET—

06 May 2013

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.

Statistics:

- **81** students took this exam.
- **67** was the median score.
- **90** was the maximum score.
- **64.05** was the average score.
- **16.34** was the standard deviation of the scores.

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?
   ◯ We were temporarily locked out of Davis 101. √ **All three Tims attended.**
   ◯ Geoff advertised a woodshop tool. ◯ We investigated a wine collection in Berkeley.

   (b) What could cause a head crash?
   ◯ Throwing a Flash drive against the wall. ◯ Tapping your fingers on your laptop keyboard. √ **Your desktop tipping over while running.** ◯ Dropping your laptop while it is off.

   (c) Many operating system crashes are caused by
   √ **device drivers.** ◯ page translation. ◯ filesystems. ◯ applications.

   (d) Which of the following is a useful approach to improving system performance?
   ◯ Choosing a benchmark randomly. ◯ Improving the parts of your code that you just know are slow. √ **Analyzing data from experiments to identify bottlenecks.** ◯ Working on the slowest part first.

   (e) When your performance data has outliers, you should
   ◯ eliminate them. ◯ assume they are experimental error. √ **understand them.** ◯ use a summary statistic.

   (f) Modern filesystems are nothing like the Berkeley Fast File System (FFS).
   ◯ True. √ **False.**

   (g) A virtual address might point to all of the following *except*
   ◯ physical memory. ◯ a disk block. ◯ a port on a hardware device.
   √ **a register on the CPU.**

   (h) Address translation allows the kernel to provide all of the following *except*
   ◯ the address space abstraction. ◯ process isolation. ◯ an inter-process communication mechanism. √ **direct access to hardware.**

   (i) Significant differences between filesystems include everything *except*
   ◯ on-disk layout. √ **reliable data storage.** ◯ data structures. ◯ crash recovery mechanisms.

   (j) Which of the following is *not* a reason that virtualization became popular?
   ◯ Difficulty migrating software setups from one machine to another. √ **Useful hardware virtualization features.** ◯ Ability to reprovision hardware resources as needed. ◯ Lack of true application isolation provided by traditional operating systems.

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) First, using appropriate terminology, describe the process of swapping out a page. For each step, explain *why* it must be performed. Second, point out how the operating system can reduce the overhead of this operation by planning ahead.

Graded by Zihe Chen.

Rubric:

- **+1 point** for each correct step.
- **+1 point** for the solution to reduce overhead.

Solution:

1. **We must remove the translation from the TLB.** This is done so that the process that owns the page cannot continue to access it while it is being written to disk.
2. **The page contents are copied to the swap file.** If they are not, then the contents cannot be preserved after the page is given to a different process.
3. **The page contents are cleared, or zeroed.** Providing previously-used pages to a process without clearing them is a security violation.
4. **The page table entry for the page is updated.** This is so the process that owns the page can locate it again the next time it is accessed.

The slow part of this operation is the disk write required to copy the contents to disk. If the operating system has a page cleaner, and the page we choose to evict has already been copied to the swap file, then it is possible we can avoid the I/O entirely.

Statistics:

- **76 out of 81** students answered this question.
- **4** was the median score.
- **3.5** was the average score.
- **1.16** was the standard deviation of the scores.

3. (5 points) List all the steps that are taken by a traditional hierarchical filesystem when creating a new file. Assume that the file is not empty.

Graded by Guru Prasad.

Rubric:

- **+1 point** for each step.

Solution:

1. Find an available inode.
2. Mark it as in use and initialize it appropriately: created time, initial size, owner, etc.
3. Find one or multiple data blocks, ideally close to the inode we just allocated.
4. Mark them as in use and link them to the inode.
5. Add an entry for the file into the appropriate directory, mapping the relative pathname to the inode number.
6. Write data.

Statistics:

- **71 out of 81** students answered this question.
- **4** was the median score.
- **3.3** was the average score.
- **1.82** was the standard deviation of the scores.

4. (5 points) Identify two tradeoffs inherent in operating system memory management. A tradeoff should involve a balance between providing two desirable properties. For each,

   - briefly describe cases in which the operating system would want to resolve the tradeoff in favor of one of the desirable properties, and
   - explain how balance is achieved during normal operation.

Graded by Zihe Chen.

Rubric:

- **+3 points** for each tradeoff. Brief descriptions were accepted.

Solution:

One tradeoff concerns page size. As the page size becomes smaller, we reduce internal fragmentation and allow finer-grained allocation and protection. However, as the page size becomes larger, we increase the amount of memory we can translate given a fixed-size translation cache (TLB). For this tradeoff, 4K seems to have emerged as a compromise solution, although there are newer systems that allow for the creation of "jumbo pages" of 32 or even 64K.

A second tradeoff is the division of memory between use for process address spaces and use as a buffer cache for one or more filesystems. Too much memory allocated to the buffer cache causes excessive swapping, while too much allocated to process address spaces causes excessive disk traffic due to buffer cache misses. We presented the Linux "swappiness" parameter which controls the runtime division of memory between the buffer cache and process address spaces. If we know that we have a very disk-heavy workload, we want as much memory for the buffer cache as possible, and "swappiness" allows Linux administrators to tune the mix.

A third tradeoff is how aggressively to clean dirty pages. If we clean frequently, we increase the probability of finding a clean page during eviction but also create a great deal of additional disk traffic. On some systems that may never be idle, we may never want to write dirty pages since the speculative disk traffic will always affect performance. If we clean too rarely, we are unlikely to locate clean pages during eviction. This interval is also tunable on many systems, including Linux.

Statistics:

- **44 out of 81** students answered this question.
- **3.5** was the median score.
- **3.7** was the average score.
- **1.32** was the standard deviation of the scores.

5. (5 points) Describe the difference between software- and hardware-managed TLBs and how they affect kernel TLB and page fault handling. Briefly describe the pros and cons of each approach.

Graded by Guru Prasad.

Rubric:

- **+2 points** for identifying differences, pros and cons.
- **+2 points** for how this affects TLB faults.
- **+1 points** for how this affects page faults.

Solution:

A software-managed TLB will generate a TLB exception if a page translation needed to resolve a virtual address is not loaded into the TLB. In contrast, a hardware-managed TLB will search the currently-loaded page tables itself to try and locate a valid translation and load it into the TLB if necessary. On a hardware-managed TLB, the kernel *never* sees TLB faults since they are resolved by hardware directly; only page faults are raised and resolved by the kernel. In contrast, a software-managed TLB sees both TLB and page faults.

The main advantage of a hardware-managed TLB is that hardware is faster at loading TLB entries and can avoid the overhead of trapping into the kernel. The disadvantage is that the kernel must implement its page tables in a way that the hardware can understand. The kernel may also lose some visibility into page usage if the hardware-managed TLB handler doesn't set some state on the page table entry to indicate that it has been faulted on and used. Advantages of the software-managed approach include more flexibility for the operating system in how to structure its page tables. That and being a better fit for operating systems coursework!

Statistics:

- **79 out of 81** students answered this question.
- **4** was the median score.
- **3.8** was the average score.
- **1.13** was the standard deviation of the scores.

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

Graded by Guru Prasad.

Rubric:

- **+2 points** for a definition.
- **+2 points** for pointing out that it helps identify the part of the system to improve.
- **+1 point** for remembering that you have to continually reassess performance as you make improvements.

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.

Statistics:

- **67 out of 81** students answered this question.
- **4** was the median score.
- **3.6** was the average score.
- **1.44** was the standard deviation of the scores.

7. (5 points) First, identify one way an instruction might fail to be "classically" virtualizable. Second, explain how this prevents the Virtual Machine Monitor from performing trap-and-emulate virtualization. Finally, briefly describe how modern virtual machine monitors work around this problem to perform full hardware virtualization.

Graded by Zihe Chen.

Rubric:

- **+2 points** for identifying how an instruction would fail to be "classically" virtualizable.
- **+2 points** for explaining how this prevents the VMM from performing trap-and-emulate.
- **+2 points** for describing how modern VMMs work around this problem.

Solution:

1. A privileged instruction is not classically virtualizable if it fails to generate an exception when run in unprivileged mode. It might just fail silently, or its behavior might be different.
2. Instructions that do not trap prevent the virtual machine monitor from receiving control and being able to check the instruction for safety and possible modify the state of the virtual machine appropriately. Essentially, if you don't trap, you can't trap-and-emulate.
3. VMware identifies these instructions before they are executed and modifies them in ways that allow it to gain control or cause the instruction to do the right thing.

Statistics:

- **21 out of 81** students answered this question.
- **4** was the median score.
- **3.5** was the average score.
- **1.56** was the standard deviation of the scores.

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**.

Both medium answer questions were graded by Guru Prasad. Unfortunately, we do not have separate statistics for each question—combined statistics for *both* medium answer questions are listed below.

Statistics:

- **13** was the median score.
- **20** was the maximum score.
- **13.0** was the average score.
- **6.04** was the standard deviation of the scores.

Identifying Identical Pages

We discussed in class how operating systems use a technique called copy-on-write to allow identical virtual pages to be shared between the parent and child process after `fork()`. Copy-on-write relies on the observation that `fork()` makes an *identical* copy of the parent's address space. So afterward, we know that for two virtual addresses in the parent and child, VA_{parent} and VA_{child}, if $VA_{parent} = VA_{child}$ at the time `fork()` is called then the pages have identical contents. Marking the shared page ready-only ensures that any modifications by either process cause a page fault allowing the kernel to create private copies of the now not-identical page.

While copy-on-write is a clever mechanism to use after `fork()`, it will not identify all possible page sharing opportunities. First, explain why not. Second, describe a system for identifying page-sharing opportunities missed by copy-on-write. Please be specific about how your system works: how it identifies identical pages, how they are merged, and how it ensures that pages that should be private stay private.

Finally, this approach emerged as a response to the increase use of virtualization technologies. Briefly explain why this is. Can you think of any other devices that would benefit from your approach?

Rubric:

- **+5 points** for explaining why COW does not identify all sharing opportunities.
- **+10 points** for a solution.
  - **+3 points** for a valid solution.
  - **+7 points** for a clear explanation.
- **+5 points** for other uses of this mechanism.

Solution:

Copy-on-write after `fork()` will only identify shared pages that emerge from the parent-child relationship. So, for example, if `/bin/whatever` forks another copy of `/bin/whatever`, then many pages may be shared via COW. However, if Alice and Bob independently start separate copies of `/bin/whatever`, then the same pages will *not* be shared.

A system for identifying shared pages that do not emerge from `fork()` has to have same way of determining that two pages are, in fact, the same. You can imagine multiple ways of doing this: computing a per-page hash, for example, which uses the page contents directly; or annotating each page with the file that it came from, which would allow identical pages from `/bin/whatever` to be identified. Hashing

might produce false positives, particularly when examining zero-fill pages that are eventually used for per-process data, and might create some overhead due to these false sharing opportunities. File annotations would not have this problem, but could not identify pages that happen to be identical even though they came from different files. That doesn't seem terribly likely though.

Using its identity function, the samepage daemon would periodically scan through pages loaded in memory. (I guess you could do ones in the swap partition too, but it's not clear how helpful that is since the goal is to reduce memory usage.) When it finds two that are identical, it do several things. First, both pages must be marked as read-only, to prevent writes during the process. Second, one of the pages has to have its PTE updated to map the virtual address to the physical address of the second page. The page should also be marked as shared between the two processes so that eviction and swap out work correctly. (Modern kernels usually have existing mechanisms to support page sharing between processes, since this can be established as a form of IPC.) Finally, the extra page can be freed. To ensure that pages stay private, we recycle the copy-on-write idea: mark them as read-only and split at the first write.

Finally, samepage merging is quite helpful for virtualization scenarios since multiple copies of the same virtual machine will share many pages but the hypervisor (or host OS) has no way to identify these pages and reduce duplicate page usage. The same approach has also been used on memory-constrained devices such as smartphones, and in particular on Android which runs many virtual machines and therefore may have many duplicated pages.

Controlling Virtual Machine Memory Usage

Virtual machine monitors typically require a substantial amount of memory in order to create a virtual machine for the guest operating system. However, when running one or multiple VMMs the host operating system may want or need to reclaim memory from them. The process of reducing guest operating system memory usage is complicated by the fact that the guest OS *thinks* that it has access to a entire large chunk of memory, the amount presented to it by the virtual machine. But in reality the host OS will want to reclaim memory from the guest OS.

In most cases the host operating system *can* directly swap out pages used by the virtual machine monitor—an approach called hypervisor (another name for VMM) swapping. However, hypervisor swapping can cause several potential problems. First, explain the two following problems in more detail, including a description of why the would occur:

- The host OS may choose the wrong page.
- Under certain conditions pages may actually be swapped in and out *twice*. (As a hint, consider what happens if the host OS has swapped out a page that is then swapped out by the guest OS.)

Second, describe a solution that allows the guest and host OS to collaborate to reduce guest OS memory usage. Keep in mind that the guest OS does not and *should not* know that it is running inside a virtual machine. Instead, you should design a way for the host OS to create memory pressure inside the guest OS and cause it to begin to evict pages itself. Describe why this is a superior approach by discussing how it addresses the weaknesses you identified above.

Rubric:

- **+5 points** for explaining the wrong page problem.
  - **+2 points** for identifying why the page could be wrong.
  - **+3 points** for understanding how and why this could occur.
  - **+1 point** for understanding the host-guest visibility problem.
- **+5 points** for explaining the double-swap problem.
  - **+3 points** for how it occurs.
  - **+2 points** for understadning the separate swap disks.
  - **-1 point** for getting the order wrong (guest first).
- **+10 points** for the balloon driver solution.
  - **+2 points** for ballooning.
  - **+2 points** for discussing how the balloon creates memory pressure in the guest.
  - **+8 points** for how the balloon works and communicates with the host.

– **+1 point** for understanding why the balloon driver needs kernel privileges in the guest operating system.

Solution:

The first part of the question asked you to identify two problems with hypervisor swapping:

- **The host OS may choose the wrong page.** This happens simply because the host OS has less visibility than the guest OS about what pages allocated to the VMM contain. It might end up swapping out part of the OS, or a page allocated to a high-priority process, or something else that shouldn't really be swapped out. (This is a not a correctness problem, however, since the page can be safely swapped in and out.) At the end of the day, the guest OS simply has more information about *what* to swap out, and so our goal is to make it make the swapping decisions.
- **Under certain conditions pages may actually be swapped in and out twice.** We provided several hints about this. Consider the following scenario: the host OS swaps out a page to its swap disk or swap file, which is distinct from the swap backing store used by the guest OS. (This would be located on the "virtual disk" used by the VMM to store the VM disk image, really just a file on the host OS filesystem.) So the page contents are copied to disk, once. Now, the guest OS decides to swap the same page out to *its* swap file. So what happens? It tries to copy the page contents to its swap file, which causes a VMM page fault and the page to be swapped *in* by the host OS. Now the guest OS copies the page contents *again* to its own swap file. Oops. We've added a round-trip back and forth to disk to what should have been a one-way flight.

So how do we solve this problem? It helps to identify what we need to do: we need a way for the host OS to create *memory pressure* inside the VMM. The guest OS will then respond to this by swapping pages out, which can be reclaimed by the host OS. The way that this is done is by installing a so-called *memory balloon driver* inside the guest OS. The balloon driver itself is under the control of the VMM or the host OS. On command, it will simply grab pages from the guest OS kernel and return them to the host OS. (It needs root privileges to do this.) However, because the balloon driver runs *inside* the guest OS, the guest OS becomes aware of the memory pressure that the host OS wants it to feel.

Essentially, you can think of the balloon as a big wasteful kernel driver that on command just chews up pages, pins them in memory (so that the guest can't try to swap them out), and reports what pages are pinned to the host OS so that they can be reclaimed from the VMM and provided to other processes. One of the reasons we need to do this, is that the guest OS is not aware of running in a virtual machine and wouldn't know what to do if the VMM suddenly changed the amount of physical

memory on the machine. Plus this would probably create contiguity issues and essentially lead to the "wrong page" problem identified above. Instead, the balloon driver communicates host OS memory requirements in a way that allows the guest OS to make the decisions about what pages to swap out.

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**.

Rubric:

Of the three long-answer questions, only one ("Deterministic Shared Memory Parallelism") was based on a specific existing system and so had at least one *specific* right answer. All three questions were intended to challenge you to think through a new problem and creatively apply some of the systems design principles we have learned this semester. In that sense they were intended to be challenging, but also hopefully fun—at least a little?

Grading for the "Deterministic Shared Memory" and "Hybrid Disk Drive" long answer questions divided into 20 points for technical correctness, divided differently for each answer (see individual question rubrics), and 5 points for quality: clarity, depth comprehensiveness, organization. Quality points are not included in the per-question rubrics and are assigned at the discretion of the grader. Aditya designed a different rubric for "Incorporating FPGAs into Kernel Designs".

Statistics:

Unfortunately, we do not have separate statistics for each long answer question. Combined statistics are below.

- **16** was the median score.
- **25** was the maximum score.
- **14.8** was the average score.
- **5.93** was the standard deviation of the scores.

Incorporating FPGAs into Kernel Designs

Field-programmable gate arrays (FPGAs) are a form of reprogrammable hardware that provide some of the flexibility of software (reprogrammability) and some of the speed of hardware, with performance much better than general-purpose processors but worse than specialized application-specific integrated circuits (ASICs). For the purposes of this question we won't get into the details of how to program FPGAs, but you should keep in mind that the speedup achieved by moving code from a general-purpose processor to an FPGA may vary based on what computations are being done.

Describe how to modify an existing operating system to effectively utilize an FPGA incorporated into a system design. Your goal should be to harness the computational horsepower of the FPGA to significantly improve application performance, while effectively multiplexing this new hardware resource. Your system should be able to decide *who* can use the FPGA but also help determine *what* they should be doing with it. There are some design tradeoffs here that you should point out and decide how to balance. You may also want to consider modifications to the executable-and-linking format (ELF) binaries we have discussed earlier this semester to support FPGA incorporation.

You can make the following assumptions:

- The FPGA cannot effectively be space-multiplexed, i.e. it can be programmed to do only one thing at a time.
- Reprogramming the FPGA takes time.
- Once the FPGA is reprogrammed, essentially a new instruction exists that can be used by any thread running on the CPU to activate the FPGA to perform a custom calculation (whatever it is programmed to do) based on inputs that could be drawn from memory or processor registers. This also implies that applications *do not* have to make a system call or involve the kernel to use the FPGA.
- The kernel has access to a program that can transform any binary code written for the native instruction-set architecture (ISA) into code to run on the FPGA, but this process has some overhead to it.

Solution: Incorporating FPGAs into Kernel Designs

Rubric:

This question was graded by Aditya Wagh.

- **+5 points** for discussing modifications to the kernel.
- **+5 points** for identifying what to do with the FPGA.
- **+5 points** for discussing how to multiplex the FPGA and policies for doing so.
- **+5 points** for identifying inherent tradeoffs.
- **+5 points** for pointing out useful modifications to the ELF format to support this architecture.

Note that you cannot use the FPGA as a TLB or general-purpose processor or memory. The question instructs you to assume that the FPGA provides *one* additional instruction, and changing what this instruction does takes time.

Solution:

There were two aspects to the answer as identified in the question: *who* should be using the FPGA, and *what* should they be doing with it? The first is about how the kernel should multiplex the FPGA resource and make it available to programs. The second is really about performance, helping applications achieve the best speedup possible.

As far as *who* should be using the FPGA, consider two of the assumptions above: the FPGA cannot be effectively space-multiplexed, and once the FPGA is reprogrammed a new special instruction exists that can be used by any running thread. Without adding any additional architectural support, these imply that only *one* application can be using the FPGA at any given point in time, and that applications must coordinate their usage with the kernel. If they do not, then application A may try to use the FPGA while it is programmed to support application B. Assuming that one "special instruction" exists, and that this instruction has no useful failure semantics (it could, and if you relaxed these assumptions your solution may differ), application A's incorrect use of the FPGA would simply (and problematically) cause the same special instruction to return a different and incorrect result. Essentially, you were provided no way for a program to determine what the FPGA is programmed to do. Instead, you need a way for the kernel to communicate to each application whether the FPGA is programmed to support it, so that the application can follow a different code path—the same code implemented for the general-purpose CPU—if the FPGA is in use by someone else.

The problem we identified above is really a subset of the general question of how applications request access to the FPGA, receive access, and have the FPGA revoked. Here's an example of an interface that could address this challenge (using OS/161-friendly types):

- `acquire_fpga(userptr_t code, bool * status, int flags)`: acquires the FPGA programmed with the transformation of the code loaded at `code`. The `flags` bitfield may include whether to block waiting for the FPGA, and whether to block again if the FPGA is preempted while this application is using it. The `status` boolean is used as a flag for the kernel to communicate whether the FPGA is programmed for this application, and can therefore can be used in *if-else* statements like this one:

```
if (status |= FPGA_FOR_ME) {
  __asm volatile("special_instruction");
} else {
  general_purpose_implementation(args);
}
```

- `revoke_fpga()`: revoke use of the FPGA. Note that this may not cause the special instruction to fail, since other applications may use the FPGA. But it can be used to cause the current application to stop blocking when the FPGA is granted to another process.

Note that reprogramming the FPGA by the same application is done by first revoking it and then re-acquiring it. The kernel may cache translated general-purpose CPU code to avoid having to retranslate code blocks each time `acquire_fpga` is called. In addition, you could have proposed that the ELF binary contain pre-translated code blocks for portions of the application that it may want to run on the FPGA. (This would, of course, couple the application both to the instruction set architecture used by the ELF and to whatever format the FPGA itself required.)

The semantics of the interface above allow an application to request *blocking* access to the FPGA. Essentially, if the FPGA is reprogrammed—perhaps for a higher-priority process—if blocking operation is requested all threads for the current application will be put to sleep until the FPGA can be restored to the state required by the application. Otherwise, the `state` shared variable is used to indicate whether the FPGA is appropriately programmed. (If blocking operation is requested, the application should never see the `state` variable set to indicate shared use.) Also note that on a multicore machine you have created a variant of TLB-shootdown, in that another core must make sure to atomicly alter the state variable and stop applications using it running on other cores before reprogramming the FPGA to ensure that it transitions safely from one application to another.

Because reprogramming the FPGA takes time, that time needs to be charged to the application. One way of doing this is simply to include the time necessary to reprogram the FPGA in the beginning of each threads scheduling quantum. In certain cases, reprogramming the FPGA simply may not make sense if the time to do so overwhelms the speedup a process will achieve.

Which is a nice segue to the performance implications. Essentially, applications want to apply Amdahl's Law to determine both *whether* it makes sense to use the FPGA and, if so, *what* for? Doing this effectively requires collecting several pieces of information for each candidate code block that could be run on the FPGA:

- How fast does it run on the general-purpose CPU? This may be a function of its inputs, but hopefully is at least somewhat predictable.
- How fast does it run on the FPGA? This allows us to determine the potential speed-up achieved *for that piece of code* by moving it to the FPGA. Determining the overall speedup requires more information, specifically . . .
- . . . how often does this piece of code run? Essentially, what is its contribution to overall performance? This could be a function of code flow, which is the whole attraction of the FPGA in the first place. However, if the process is going to take the hit of reprogramming the FPGA, it probably wants to be fairly sure that the code will run often enough in the future to justify the programming time.
- How long does it take to reprogram the FPGA? You could include the translation time, but that's really only a one-time cost and could be pre-computed. You could also have argued that reprogramming time is constant and doesn't depend on what the FPGA is being programmed to do, which it might be.

These are some of the performance issues we wanted you to raise. Kernels already provide various profiling tools to allow applications to understand their performance on general-purpose CPUs. You'd need some extension of this to allow the application to determine how long the special instruction takes when it runs. That's really all that's required in terms of new features. Since we are charging the application for the programming time, if it uses it poorly it will simply slow itself down.

Deterministic Shared Memory Parallelism

(Credit for this question goes to Bryan Ford at Yale and his Determinator system.)

Earlier this semester we presented synchronization primitives as a solution to *race conditions*, which we defined as cases where the ordering and timing of thread execution would cause a result to be incorrect or unpredictable. (Remember the bank account example?) Reasoning about how threads might interleave at runtime and the pattern of their accesses to shared memory has made multi-threaded programming difficult for years.

Part of the problem is that, when accessing shared memory, reads and writes from each thread happen synchronously, meaning that if synchronization primitives are not used correctly threads can observe variables in an inconsistent state. Consider the following game-based scenario. During each time step, each actor in the game performs some action and updates its position based on the position of all other actors. The game uses a separate thread to perform the computation for each actor, but all threads read from and write to a state table stored in shared memory. However, the position of an actor is defined as a tuple in three-dimensional space, (x, y, z), and so requires three memory writes to change and three memory reads to access, and so absent proper synchronization is *not* atomic.

The next page contains pseudo-code for the game described above. `thread_fork` you are familiar with: it creates a new thread, in this case a new *user thread*. The `thread_join` command will wait for the thread to complete before returning. It's analogous to `waitpid` but for threads, rather than processes.

First, describe how a race condition can occur if the table is not properly synchronized. You should identify a condition where one thread will observe a shared variable in an inconsistent state. Briefly, outline a solution to this problem that uses one of the synchronization primitives we discussed in class. (Clearly, this is not the *real* question.)

Second, obviously the world would be a better place if every programmer used synchronization primitives correctly. But the world is not that place, sadly, and so operating systems may want to aid programmers by providing primitives that improve safety during multithreading. Propose a solution to the race condition you identified above that *does not* modify the application code above. Instead, you should confine your modifications to `thread_fork` and `thread_join`. As a hint, you might want to consider the design and operation of the Git version control system that you have been using throughout the semester.

```
 1
 2 void update_actor(int i) {
 3   // ... examine state of other actors ...
 4   state[i].x = new_x
 5   state[i].y = new_y
 6   state[i].z = new_z
 7 }
 8
 9 int main() {
10   // ... initialize state of all actors ...
11   for (int time = 0; ; time++) {
12     thread t[NACTORS];
13     for (i = 0; i < NACTORS; i++) {
14       t[i] = thread_fork(update_actor, i);
15     }
16     for (i = 0; i < NACTORS; i++) {
17       thread_join(t[i]);
18     }
19   }
20 }
```

Solution: Deterministic Shared Memory Parallelism

Rubric:

This question was graded by Guru Prasad.

- **+5 points** for identifying and fixing the race condition using synchronization primitives.
  - **+3 points** for identify the race condition.
  - **+2 points** for solving it with a synchronization primitive.
  - **+2 points** for clarity.
- **+15 points** for the Determinator solution.
  - **+5 points** for understanding the Git analogy.
  - **+12 points** for the solution.
  - **+3 points** for what to do with merge conflicts.
  - **+3 points** for clarity.
- **+5 points** for clarity and correctness. (See previous discussion of long answer rubrics.)

Solution:

There were two parts to this question. The first asked you to identify a race condition in the provided code and address it using a synchronization primitive. The second asked you to fix this race *and* some amount of additional synchronization problems without modifying the code, but instead by modifying `thread_fork` and `thread_join`.

First, an example of the race that can occur is the following:

- **Thread i**: `state[i].x = new_x`
- **Thread i**: `state[i].y = new_y`
- (. . . note that `state[i]` is now in an inconsistent state . . .)
- **Thread j**: `// ...examine state of other actors ...`

At this point, Thread j will observe `state[i]` in an inconsistent state. Solving this is straightforward: we add a lock and use it to protect the state array during updates and examination:

```
struct lock * state_lock;

void update_actor(int i) {
  lock_acquire(state_lock);
  // ... examine state of other actors ...
  state[i].x = new_x
```

```
    state[i].y = new_y
    state[i].z = new_z
    lock_release(state_lock);
}
```

Note one drawback of this solution: grabbing the lock essentially serializes the `update_actor` function, which is largely unnecessary. If we assume that `update_actor` runs during each time step and what we *want* to happen is it to observe the *last* state of each thread $j \neq i$ for Thread i and then update `state[i]`, then it is safe to interleave all of the reads if we can improve the handling of writes (or updates).

The hint asked you to consider the operation of Git, the distributed version control system we have been using this system. Quite a good analogy, we think, and a very good hint. Consider the difference between concurrent editing of your OS/161 source code—either on the same machine in different terminals, or using Dropbox—and compare it with using Git. What's different with Git? Two things:

1. Git makes merges *explicit* and makes them happen at well-defined points in time defined by each user—or thread, in this case. Until you call `git merge`, you only see the changes you have made locally. (This happens implicitly when you use `git pull`, in case you are wondering why you've never used `git merge` explicitly all semester. `git pull` is roughly a `git fetch` followed by a `git merge`.) When you use Git, you only see your partners changes when you are ready; if you share files directly, you see them whenever they happen to perform a write, which now demands more careful synchronization. (People who use Dropbox improperly to work on collaborative documents concoct all kinds of gross schemes involving explicit locking that make Git users chuckle.)
2. In many cases, Git is able to merge multiple files even if they have been modified concurrently, as long as the changes don't overlap or conflict with each other. Clearly, the structure of source code makes this easier; text files and Latex also work reasonably well. Merging files with binary formats, such as PDFs or Microsoft Word documents is hard or impossible.

The application of these ideas to the problem you were given is fairly straightforward:

1. `thread_fork` performs the equivalent of a `git clone` —or what is sometimes called a *checkout* on other version control systems—but on the processes *address space*. Until it calls `thread_join`, changes to memory made by Thread i are not seen by any other thread; it has its own private copy of memory.
2. `thread_join` performs the equivalent of a `git merge`, but on the processes address space. When Thread i calls `thread_join`, only the parts of memory it has changed are merged back into the public address space to be seen by the main thread and other threads that call `thread_fork` in the future.

Note that in many cases merging changes to memory are even easier than merging changes to files. Why? Because memory doesn't have the equivalent of an append: you can't add changes into the middle of a structure the way you can when editing files. (Or maybe that it's just not done that way, not that it's impossible.) We can write some simple pseudo-code for the function that merges the memory regions during `thread_join`:

```
for all valid process addresses:
  if original_address.value != thread_address.value:
    original_address.value = thread_address.value
```

What would happen to our (un-modified) code above? For each Thread i, the only portion of memory that it would have modified would be its portion of the `state` table, and all the modifications would be compatible. However, we do need to consider what happens if the merge fails. First, we need to determine how to detect merge failure: it's probably as simple as maintaining state for each block delineated by `thread_fork` and `thread_join`. More pseudo-code:

```
thread_fork:
  for all valid process address:
    original_address.modified = False

thread_join:
  for all valid process addresses:
    if original_address.value != thread_address.value:
      if original_address.modified:
        // what should we do?
      else:
        original_address.value = thread_address.value
        original_address.modified = True
```

What the Determinator system—which this question is based on—does is treat merge conflicts as a software exception, which will crash and end the process similar to a divide-by-zero error. That may be the best thing you can do here, but it's a nicer way to discover a concurrency violation than a race condition that could produce a serious error or just give GWA too much moolah.

Characteristic	SSD	HDD
Price-per-gigabyte	$0.59	$0.05 (See note 1.)
Random access time	100 μs	10 ms
Data transfer rate	100 MB/s	100 MB/s (See note 2.)
Durability	No moving parts, resistant to shock and vibration	Sensitive to shock and vibration

[1] For the 3.5 in form factor.

[2] Once the head is positioned.

Table 1: Comparison of HDD and SSD disk drives.

Hybrid Disk Drives

Today disks are at a crossroads. Magnetic spinning drives still offer a order-of-magnitude improvement in price-per-byte, but Flash technologies are improving latencies and bandwidth. Specifically, you can make the following assumptions about magnetic hard-disk drives (HDDs) and Flash-based solid-state drives (SSDs) listed in Table 1.

Your new company, HyDrive ®, thinks it can provide the best of both worlds by shipping a hybrid hard drive *and* new filesystem. Your hybrid drive contains both a large(r) HDD and a small(er) SSD, each of which is block-addressed and independently controlled; consider them two independent disks that are sold inside the same package. Unfortunately, *everyone*—HyDisk ®, DriveBrid ®, DiskBrid ®, Grapes ®—is cramming two disks into one box and pretending that it's innovative. What sets HyDrive apart is the innovative new filesystem you are selling that takes full advantage of the capabilities of both the HDD and SSD.

Describe the design of the breakthrough HyDrive ®filesystem. You can use any of the ideas presented in class, but should aim to achieve the best of both worlds and have a single filesystem spanning both disks, with seek times as fast (or close to) as the SSD, capacity as large as (or close to) the HDD and SSD combined, and better durability and failure recovery than an HDD alone. Be very specific about the design and implementation of your new filesystem. In particular, it is necessary (but not sufficient) to address the following questions:

- What file metadata is required to name files and locate their contents?
- Where is file metadata stored?
- Where is file data stored?
- How does your filesystem determine where to put file metadata and file data?
- How does your filesystem recover after an unplanned power outage or reboot? How do you recover from failures on the HDD? What about failures on the SSD?
- How does your design harness the best features of the SSD? What about the benefits of the HDD?

Keep in mind that because you are implementing a *filesystem*, rather than a disk controller, you have visibility into file operations such as reads and writes and don't have to confine yourself to observing block-level operations.

Solution: Hybrid Disk Drives

Rubric:

This question was graded by Aditya Wagh.

- **+10 points** for answering first five questions, with two points each.
- **+10 points** for addressing the last question using the information provided.
- **+5 points** for clarity and correctness. (See previous discussion of long answer rubrics.)

Note that it was particularly important to notice the similar transfer rates but differing seek times, which allows you to do better than simply use the SSD as a cache. Also note that we attempted to award points for answering the first five questions even if that answer was embedded somewhere in your solution.

Solution:

There were really two salient differences and one similarity between the two drives that we were hoping you would notice and address as part of your solution:

- Difference: random access is *two orders of magnitude* better on the SSD.
- Difference: the SSD is more resistant to shock and vibration than the HDD.
- Similarity: once the seek is completed, the data transfer rates of the SSD and HDD are *the same*. (This isn't quite true since we provided numbers for slower SSDs and faster HDDs, but it was the assumption we allowed you to make.)

(Note that the cost difference has already been factored into the design of the Hy-Drive ®hardware, since it integrates a small(er) SSD and large(r) HDD.)

The simplest—but incomplete—solution was to use the SSD as a block cache for the HDD. The approach would be to put the most commonly-used blocks on the SSD and the rest on the HDD. If the SSD is a true cache, then all blocks are always located on the HDD and simply copied onto the SSD when they are deemed "hot" enough. Note, however, that this solution does not satisfy our capacity goal, since the capacity of the hybrid drive with the SSD acting *only* as a cache would be only the capacity of the HDD alone. A modification to this solution that improves capacity would be to migrate blocks back-and-forth between the HDD and the SSD as needed, but clearly this creates more load on both disks to perform the copies and could interfere with performance.

But we wanted you to do better than the simple block-cache solution, which doesn't exploit the fact that you are developing a *filesystem* that will be tailored to this hybrid device. Everything we have discussed so far does not rely on or leverage the extra visibility that filesystems have, and could in fact be implemented as part of the buffer

cache below the filesystem layer. In addition, this solution does not address fault tolerance, since you have to hope that when the head crashes on the HDD, the blocks you need to perform recovery are on the SSD. Finally, note that we asked to you assume that *once the head is positioned* the data transfer rates are equivalent, a fact that the block-cache solution does not exploit. Let's do better.

First, let's think about where to put filesystem metadata. The answer to the first question ("What file metadata is required to name files and locate their contents?") could really be just the usual stuff typical to many hierarchical filesystems: superblocks, data blocks, inodes, and free inode and data block bitmaps. For fault tolerance, the critical parts of the filesystems on-disk data structures might be better placed on the SSD, since it is less likely to fail mechanically (sector failures) or catastrophically (head crash).

As far as where to store file metadata and data (the second and third questions), this is where things get more interesting. Remember that you were told to assume that the data transfer rate for the SSD and HDD was *the same*, and only the seek times differ. So let's think about what we want to be happening under heavy load. One way is to try and preserve an overall transfer rate of 100 MB/s by using the SSD to hide seek latencies on the HDD. So the joint operation of the disk could look something like this:

- **t = 0**: initiate seek for both SSD and HDD.
- **t = 100 μs**: SSD seek completes. Begin reading data at 100 MB/s.
- **t = 10 ms**: HDD seek completes. Begin reading data at 100 MB/s, disable SSD read.
- **t = 100 ms - 100μs:** HDD read about to complete, begin SSD seek.
- **t = 100 ms**: HDD read completes, SSD seek ends, begin SSD read, begin next HDD seek.
- . . . continue in this way to preserve 100 MB/s transfer rate . . .

By dovetailing the reads and writes carefully, we can create the illusion that the disk is both (a) as big as the HDD and SSD combined but (b) without any read latency and capable of *sustained* reads (or writes) at 100 MB/s. So this solution potentially combines two of the desirable features of the SSD (low seek latency) and HDD (size).

Sounds great, but how do we implement this? Dovetailing the I/O operations properly requires that we distribute file metadata and data carefully between the SSD and the HDD. On the HDD all of the usual locality tricks apply, since it is still a spinning platter with travel between different blocks dominated by the head seek latency. What is interesting here and new is determining how to distribute file content between the two drives.

One potential approach is to try and place as much file metadata as possible on the SSD, since we've noted that many file operations require updating the inode. So while

we are seeking the HDD to the location of the data blocks we need from the file, we can be making whatever changes to the metadata need to be made on the SSD.

If this is not sufficient to hide the seek latencies—and it may not be, given that the HDD seek latency is 100 times larger—another approach is to locate the first N data blocks of the file on the SSD near the inode with the rest on the HDD. This approach works best for files that are read (or written) sequentially, since we are hoping that while we are retrieving the first few data blocks from the SSD the HDD has time to seek to the next block.

This refinement points to another potential way of trying to place data blocks: the SSD is better at supporting random access, so we might want to put not the most often accessed files but the most *randomly* accessed files on the SSD, since they will benefit most from speedy seeks. Files that are mainly accessed sequentially should achieve good performance on the HDD with traditional data location techniques since the long seek is amortized over the rest of the sequential read. (We might still put the first blocks and the inode on the SSD, however.) Layout techniques from existing hierarchical filesystems could be adapted and made to make even more accommodations for sequential access, since random I/Os should be soaked up by the SSD. Capacity becomes a concern here, however, since the SSD may be best used for files that are both randomly-accessed and *small*, given its reduced capacity (due to its higher price-per-byte).

Finally, note that our attempt to preserve a continuous 100 MB/s transfer rate actually understates what the combined device can do. Assuming the interconnect to the rest of the machine can support it, we can do *200 MB/s* once the HDD heads are positioned by reading/writing from both disks simultaneously. This leads to yet another idea for file placement: by desired transfer rate. Files that do not require rapid transfers, such as media files that are usually accessed at steady and fixed rates, can go on the HDD where they are affected by seek times. Files that need more consistent transfer rates can go on the SSD and exploit its low seek times. And files that need *extremely* high transfer rates can be split between the SSD and HDD where they can potentially achieve closer to 200 MB/s. Note that this would require either new profiling techniques to determine for which files transfer rate affects system performance, or the addition of file "priorities" to allow users to prioritize some files over others. (Sounds like an interesting idea!)

With respect to failure and recovery, our main goal was for you to notice that the SSD is generally more durable and would be a good place to put your important filesystem data structures (inodes, allocation bitmaps, superblocks) and recovery structures (journal). While parts of the HDD may fail or not survive a catastrophic event, hopefully the SSD preserves enough information to recover as much data from the combined device as possible. Similarly, duplicating important metadata on the HDD (which is larger anyway) will help reduce the probability that a SSD failure will render the drive unusable. It is likely that a HDD failure would cause data loss, which the

user would notice; in certain cases, a SSD failure might *not* cause data loss but would affect performance and degrade the survivability of the drive, so you would want to notify the user at that point even if the drive continues to seem functional.

We've presented a few potential starting points for complete solutions, but this question was intended to be quite open-ended. What did you come up with?

Finally, note that we did *not* ask you to deal with two of the more thorny aspects of SSD data placement. First, Flash drives have "erase units" that are usually larger than typical file blocks, meaning that in order to write one byte you first have read 32 K, erase it, and the rewrite the modified content. Second, Flash drives also wear out, with each erase unit only being able to be erased and rewritten some number of times. This property creates the need for "wear leveling" to try and allow the entire drive to fail at once, instead of some portions failing much faster. However, with a hybrid drive you might not care as much about wear leveling and instead allow the drive to simply lose capacity over time, as long as you could migrate content to the HDD and simply sacrifice performance. In any case, you weren't required to consider these complications.