CSE 421/521 Final Exam Solutions

—SOLUTION SET—

12 May 2014

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 (very) 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. The point value assigned to each question is intended to suggest how to allocate your time. **No aids of any kind are permitted.**

Statistics:

- **147** students took this exam.
- **75** was the median score.
- **72.33** was the average score.
- **18.71** 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) The funniest thing that happened *during* class this semester was when
   ◯ Guru fell asleep. √ **someone thought that one class was actually only five minutes long.** √ **Geoff discussed what to do if you are flying a helicopter and someone points a laser pointer at you**[1]**.** √ **a PDF rotated unexpectedly.**

   (b) Which of the following is *not* a difference between the Xen and VMware approaches to virtualization?
   ◯ Approach to instructions that are not classically virtualizable √ **Support for the x86 architecture** ◯ Ability to run unmodified guest OSes ◯ The way that guest OSes modify their page tables

   (c) It is the *most* difficult to get repeatable performance results when
   ◯ consulting Jinghao's blog. ◯ running a system simulator. √ **measuring a real system.** ◯ using a system model.

   (d) Which of the following is *not* a page replacement algorithm we discussed?
   √ **FIFO** ◯ Least-Recently Used (LRU) ◯ Clock ◯ The Oracle

   (e) All of the following are RAID levels presented by Patterson et. al's paper *except*
   √ **RAID 0.** ◯ RAID 1. ◯ RAID 4. ◯ RAID 5.

   (f) Which of the following is a requirement of implementing a virtual machine?
   ◯ Synchronicity √ **Performance** ◯ Atomicity ◯ Carl Nuessle

   (g) A correctly-implemented journaling filesystem will never lose data.
   ◯ True ◯ Only if Zihe implemented it √ **False**

   (h) Which was one motivation for log-structured filesystems?
   ◯ Disks were getting faster. √ **Filesystem buffer caches were getting larger.** ◯ John Gerber's 80s hairstyle ◯ Filesystem workloads were increasingly dominated by writes.

   (i) Which is *not* one of Butler Lampson's hints for improving system performance?
   ◯ Use hints. ◯ Cache answers. ◯ Compute in the background. √ **Aggregate load.**

   (j) Which is most likely to cause a disk head crash?
   ◯ Scott Haseley's music collection ◯ ASST4 √ **Dropping a running laptop off the roof of Davis** ◯ Your smartphone falling out of your pocket

[1] Just continue flying the helicopter! Simple.

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) Now that you've implemented the OS/161 system calls, let's improve their performance. Might as well start with `getpid()`, since it's simple. **First**, briefly explain the overhead of performing a call to `getpid()` (1 point). **Second**, propose a way to reduce this overhead by exploiting properties of the `getpid()` system call (2 points). **Finally**, will this improve performance? If so, why? If not, why not? Justify your answer (2 points).

Graded by Guru Prasad.

Rubric:

- **+1 point** for identifying the overhead
- **+2 points** for reducing the overhead
- **+2 points** for identifying that this will not reduce performance

Solution:

The overhead of `getpid()` is usual user-kernel transition overhead involving saving state on the way into the kernel, incurred by every system call. The way to avoid this is to note that each process ID is set during `fork()` (or `clone()`, if you are supporting threads) and doesn't change during the life of the process. So there is no need for the process to enter the kernel repeatedly when asking for it. Instead, the C library could simply cache the answer, or the kernel could write a copy of the process ID to an agree-upon memory location during `fork()` and then `getpid()` could be modified to retrieve it using a single load, which would be much more efficient that trapping into the kernel.

Will this improve overall process performance? Of course not. Apply Amdahl's Law. Most processes rarely if ever retrieve their process ID, and if they do they probably cache it using a local (or global) variable for the same reasons we just described. Or, more colloquially, as Adam put it: "Processes just don't call `getpid()` in a loop!" Hopefully not.

Note that (a) this question was released to the class before the exam, and (b) it turns out that it (unintentionally) also appeared on the 2012 Alternate Midterm. So hopefully people did well on this question!

Statistics:

- **143 out of 147** students answered this question.
- **3** was the median score.

- **3.15** was the average score.
- **1.55** was the standard deviation of the scores.

3. (5 points) Google's File System (GFS) and RAID share a common approach to improving performance. **First**, describe the approach (3 points). **Second**, identify one pro (1 point) and one con (1 point) with this technique.

Graded by Zihe Chen.

Rubric:

- **+3 points** for describing the approach
- **+1 point** each for one pro and one con

Solution:

The shared approach is to build high-performance systems using large amounts of commodity hardware. In the case of RAID, the goal was to replace a single expensive disk with multiple cheaper disks. In the case of GFS, the goal was to replace a smaller number of expensive servers with a larger number of cheaper commodity machines.

In both cases one of the key pros is that the price-performance curve shallows as you go up; or, put another way, that low-end machines give you more "bang for your buck" than high-end machines. So you can buy more disk bandwidth or server bandwidth for the same amount of money. This is really the most important benefit.

The main drawback is by using more of anything—whether its disks or servers—to accomplish the same task you degrade reliability. Given any non-zero probability that a component will fail in a given period of time, the probability that N components will fail approaches 1 as N gets larger. So you have to deal with the reliability problems you've created. This is really the most important drawback.

Statistics:

- **79 out of 147** students answered this question.
- **4** was the median score.
- **3.15** was the average score.
- **1.66** was the standard deviation of the scores.

4. (5 points) Describe two ways that filesystems compensate for or are designed around specific properties of spinning disks. For each, **first** identify what the property is (1 point) and **second** explain how the filesystem is designed around it (2 points).

Graded by Zihe Chen.

Rubric:

- **+1 point** per property identified
- **+2 points** per filesystem design exploit or workaround

Up to 5 points total.

Solution:

There are a bunch of these. Here's a non-exhaustive list:

- **Data transfer limitations.** FFS worked around bus-speed limitations of early disks by not writing data to consecutive blocks, since doing so would cause the disk buffer to fill and force it to stall and make a complete rotation before it could continue reading. Instead, it wrote files to alternate blocks at intervals allowing it to match the disk transfer speed.
- **Rotational effects.** We mentioned that many filesystems try to put frequently-accessed content at the outer edge of the disk—or at least start filling the disk at that edge—because constant disk density combined with longer tracks on the outer edge makes data transfer higher to and from that part of the disk.
- **Seek times.** We discussed many different attempts to reduce the impact of seek times, including (1) utilizing a buffer cache, (2) locating inodes and file content close to each other, (3) locating data blocks from the same file close to each other, and (4) locating related files close to each other—where distance is measured on the disk in terms of the time needed to seek the heads from one track to another.
- **Cylinder groups.** FFS also used *cylinder groups* to locate related files and file metadata close to each other on the disk, specifically at places where they could be read and written from multiple platters without moving the heads (very far). This exploits the fact that disks are created of multiple platters but the disk arm positions all heads in the same location at the same time.

  The idea of effectively breaking the filesystem into smaller filesystems, each with its own inodes and data blocks, is preserved by modern filesystems such as `ext4`. The goal is the same: reduce seeks between file metadata (i.e., inodes) and file content (i.e., data blocks) by structuring the file system so that they are nearby on disk.
- **Log-structured filesystems.** LFS tried to address seek time limitations in a different way by performing as many disk operations as possible to one location on the disk: the end of the append-only log. This works as long as a large buffer cache soaks up all or most reads, leaving only writes to be performed by the disk.

- **Journaling.** Works around disk reliability problems by exploiting the atomicity of operations to single disk blocks. If the filesystem can succinctly describe what it was attempting to do when that operation spans multiple disk blocks, then it is easier to identify and either complete or abort unfinished operations when recovering from a sudden failure.

I probably missed a few that you remembered.

Statistics:

- **134 out of 147** students answered this question.
- **5** was the median score. (This question was too easy.)
- **3.98** was the average score.
- **1.43** was the standard deviation of the scores.

5. (5 points) Describe each step in the process of translating the path `/home/trinity/os161` to an inode number in an FFS-like filesystem (1 point each step).

Graded by Zihe Chen.

Rubric:

- **+1 point** per step correctly identified.

Up to 5 points total.

Solution:

This was drawn pretty much exactly from the lecture slides.

1. Use the hardcoded value to map the root path component to an inode number. Let's say this is 2.
2. Open the directory with inode number 2.
3. Look for the "home" pathname component in directory 2 and map it to the next inode number, say 3.
4. Open the directory with inode number 3.
5. Look for the "trinity" pathname component in directory 3 and map it to the next inode number, say 4.
6. Open the directory with inode number 4.
7. Look for the "os161" pathname component in directory 4 and map it to the final inode number, say 5.

If at any point one of the pathnames was *not* a directory, the mapping would fail.

Statistics:

- **125 out of 147** students answered this question.
- **5** was the median score. (This question was too easy.)
- **4.22** was the average score.
- **1.31** was the standard deviation of the scores.

6. (5 points) Consider the lock interface used by OS/161. **First**, give an example of improper use of locks that violates the lock interface (2 points). **Second**, explain how good interfaces minimize the disruption required to improve existing code (3 points).

Graded by Zihe Chen.

Rubric:

- **+2 points** for an improper use of the lock interface
- **+3 points** for describing how good interfaces minimize the disruption caused by improving existing code

Solution:

Here are a few different ways to misuse the lock interface:

- Acquiring the same lock you already hold.
- Freeing a lock that you do not hold.
- Call `lock_do_i_hold()`, which is not part of the public lock interface.
- Poke around in the internal structure of the lock, inspecting or manually changing the owner, etc.

We were really hoping that you would identify the second pair more than the first, the reason being that the lock interface can defend itself from improper acquires and releases, but not from improper use of what are supposed to be private methods or internal tampering—this isn't Java, and one of the nice things about Java is its rigidity about defending interface definitions.

The reason why good interfaces and good interface *usage* simplify improving existing code is that they allow programmers to change the implementations of their functions without worrying about breaking other things. In many cases, library maintainers not only didn't write but can't even imagine all the code that is using their functionality—consider the C library as an example, and how many millions or billions of lines of code depend on that library. So the interface description represents a contract between the functions implementer and its users. The users agree to use the function a certain way, and the implementer agrees that the interface will provide certain features.

In low-level languages like C interfaces are more a matter of convention and aren't enforced at the language level, but are still equally important. Consider the lock example above. Because `lock_do_i_hold()` isn't part of the public interface, it isn't guaranteed to (a) work the same way over time or (b) even exist from generation to generation as the lock code changes. So if your code improperly uses and depends on it, you may be in for a nasty surprise one day when entire chunks of your previously-working program stop functioning.

Statistics:

- **92 out of 147** students answered this question.
- **2** was the median score. (This question was hard.)
- **2.45** was the average score.
- **1.8** was the standard deviation of the scores.

7. (5 points) The `POPF` x86 instruction is intended to alter the interrupt flag (`IF`) system flag bit, which controls which interrupts the processor will handle[2]. However, if `POPF` is executed by an unprivileged process it simply fails silently, i.e. it is executed but does not modify the interrupt flag.

   First, describe why `POPF` makes x86 difficult to virtualize (1 point). What property is it missing that would make virtualization easier? **Second**, briefly describe how use of `POPF` by the guest OS would be handled by full hardware virtualization (such as VMware) and by paravirtualization (such as Xen) (2 points each).

Graded by Guru Prasad.

Rubric:

- **+1 point** for identifying the virtualization challenge posed by `POPF`
- **+2 points** for describing how this would be handled by full hardware virtualization approaches
- **+2 points** for describing how this would be handled by paravirtualization approaches

Solution:

`POPF` is not a classically-virtualizable instruction. Instead of trapping when run in unprivileged mode, which is what we need to perform trap-and-emulate virtualization, it just fails silently. Alternatively, it behaves differently when run as an unprivileged user rather than raising an exception. So if we are trying to run a guest OS at a lowered privilege level, there is no mechanism to alert us when it tries to execute this instruction and it does not have the intended effect.

Full hardware virtualization approaches, such as VMware, can't change the guest OS. So they have to detect that the `POPF` instruction is about to be executed and rewrite it to a safe series of instructions that either (a) has the intended effect directly or (b) enters the VMM so that it can adjust the state of the VM `IF` register.

Paravirtualization approaches, such as Xen, can change the guest OS. So they would rewrite any instruction sequences that used `POPF` to instead either (a) trap into the VMM or (b) utilize a new VMM interface designed to support altering the `IF` register.

Statistics:

- **83 out of 147** students answered this question.
- **3** was the median score. (This question was too easy.)
- **2.46** was the average score.
- **1.72** was the standard deviation of the scores.

[2] So it acts somewhat similarly to the `splx()` command in OS/161, although at the hardware level.

Medium Answer

Choose 1 of the following 2 questions to answer. **Do not answer both questions.** If you do, we will only read the shorter one. Complete this question on attached scratch paper. Clearly label your answer.

8. (20 points) Filesystem Versioning

Versioning is an attractive feature for filesystems to provide. It means that not only does the filesystem provide access to the most recent file content through the familiar API, but also uses available disk space to store older version of files and directories as well as deleted files and directories.

Let's consider how to add this feature to the traditional hierarchical filesystem designs we've discussed. Here are design requirements for Our Versioned Filesystem (OVFS):

1. OVFS does not track every change to every file. Instead, it creates periodic *snapshots* that record the state of the filesystem at a particular point in time.
2. OVFS provides access to old file versions without changing the familiar UNIX file API—through `open()`, `close()`, `read()`—with the exception that old file versions cannot be written to, only read from. (Attempts to open file versions with write permissions or perform writes can fail with an error.)
3. OVFS utilizes all available space to store old file versions. Once the disk capacity is reached, it should apply a reasonable policy to remove old versions and create space for new file content.

Present a design for OVFS. **First**, describe changes to filesystem naming required to support #2 without causing undue interference to users' abilities to name files normally (5 points). **Second**, describe how to perform a snapshot to store old file versions (5 points). Keep in mind that other file activity may be taking place, and consider whether you can provide a coherent snapshot of the entire filesystem at an exact moment in time or something more approximate. **Third**, present a default policy for removing old file versions and argue why it is an appropriate default (5 points). **Finally**, describe one performance or storage optimization that you can implement in OVFS to either reduce the disk activity or storage overhead of file versioning compared with the simplest approach (5 points).

Graded by Jinghao Shi.

Rubric:

- **+5 points** for describing the naming changes required to support versioning
- **+5 points** for describing how to perform a snapshot
- **+5 points** for describing a default policy for removing old versions
- **+5 points** for describing one optimization to reduce disk activity or storage overhead (or both)

Solution:

If you've used a versioning filesystem previously, you probably had a leg up on this question. Let's start by discussing the naming issue. The question asked you to provide access to old versions without changing the familiar UNIX file API—meaning that they need to be part of the rest of the file namespace—but also specified that you should do this *without* causing undue interference to users' abilities to name files normally. So you don't want to version `/path/to/my/file` on May 13th, 2014, as `/path/to/my/file.2014.05.13`. Instead, the common approach is to use *hidden directories* to store snapshots containing versioned files. These can be placed anywhere, as long as the directory structure under them matches the filesystem they are versioning. For example, if we call our snapshot directories `.snapshot`, we could version `/path/to/my/file` into `/.snapshot/2014/05/13/path/to/my/file` or `/path/to/my/.snapshot/2014/05/13/file`. Placing snapshot directories closer to the leaves of the filesystem makes them more difficult to browse, but also makes it more likely that they store only files used by a single user on a multi-user system. Many versioning filesystems on shared-user systems place a snapshot in each users home directory as a nice compromise, so `/home/trinity/os161.conf` goes into `/home/trinity/.snapshot/2014/05/13/os161.conf`.

Performing a snapshot is conceptually quite easy: simply copy all files into the appropriate subdirectory in the `.snapshot` directory. For example, on May 13th, 2014, you would execute the equivalent of a

```
$ cp -aR /home/trinity /home/trinity/.snapshot/2014/05/13
```

However, as hinted this is complicated by the fact that ongoing file operations may make it difficult to provide a coherent snapshot actually representing the *exact* state of every file at that exact moment. Easiest way to deal with this? Don't. Hopefully you didn't try to! Approximate snapshots are fine, although they may not correctly preserve interfile dependencies.

At some point you're going to start running out of space. There are a variety of good policies for pruning old versions. One approach is to keep a lot of recent versions, say every 15 minutes, since those soak up a lot of mistakes: "Did I really just delete that entire directory?" Alongside those, you keep a sparser set of old versions, say one per month, to aid with the "Where is that copy of my 2010 tax return?" type of problems.

Finally, there are a lot of ways to reduce overhead. The main thing to observe is that most files don't change between every version, so there is no need to copy them. (You could identify them using standard file access timestamps.) The only concern this raises, of course, is interaction with your version pruning approach, since now not every version will contain all files. A simple solution is to perform two kinds of snapshots: incremental snapshots, which are low-overhead and can be performed frequently; and full snapshots, which are high-overhead and should not be performed frequently. Then you can delete incremental snapshots knowing that you will never lose file versions past the point of the previous full snapshot.

Note that this question was released to the class before the exam.

Statistics:

- **138 out of 147** students answered this question.
- **18** was the median score.
- **16.67** was the average score.
- **3.03** was the standard deviation of the scores.

9. (20 points) Improving `exec()` ELF Loading

Recall that the ELF binary file contains an address space blueprint used by `exec()`. Depending on the process, the executable file can be quite large.

The simplest approach to implementing `exec()` is to have the ELF loader issue writes to load all of the code pages from the ELF executable file. Each page load will generate a VM fault which the OS will handle by adding pages to the code region allowing the content to be copied from the file into the processes address space. The result is that when the process begins running, all of its code pages are already present. OS/161's ELF loader does this by default.

Unfortunately, this violates one of our core system design principles! Let's address the problem[3]. **First**, identify the system design principle that is being violated and describe how (5 points). **Second**, describe a way to modify the ELF load during `exec()` to avoid this problem (10 points). Your solution should *completely* eliminate disk activity during `exec()`, at the price of increased activity as the process begins to execute. Carefully describe what happens during `exec()`, what happens afterward, and any changes to core paging data structures that are required by your approach. **Third**, identify one complication with this approach that could cause processes to crash and describe a way to address it (5 points).

Graded by Jinghao Shi.

Rubric:

- **+5 points** for identifying the systems design principles that is being violated
- **+10 points** for a solution
- **+5 points for identifying and addressing the complication**

Solution:

To begin, the design principle that is being violated is the idea that procrastination can be useful. We discussed on-demand paging in class, and this is the opposite: we are loading a great deal of code into the process's address space that it will probably never use. Granted much of this will end up swapped out once we are under memory pressure, but it still creates a lot of disk activity and unnecessary memory usage to load it in to the address space in the first place—and then to swap it out later. So let's fix that.

Note that our goal is to eliminate *all* disk activity during `exec()`, meaning that just writing the pages directly to swap isn't an option. That actually creates a lot of disk activity, roughly twice of the normal amount, since I have to (1) read pages from disk and (2) write them to swap. Clearly, the requirement to completely eliminate disk

[3] You do not need to consider shared libraries—assume that all code needed by the process is in its own ELF executable, either because it does not use shared libraries or was statically linked.

activity during `exec()` means that we have to leave content where it is—in the ELF binary itself.

Recall that there are two kinds of content that get loaded into the address space during `exec()`: code, and statically-initialized data. (Any data that gets initialized to zero simply gets marked as zero-filled and pages are added as needed, so that's already doing what we want.) So instead of loading these into the address space, we need to mark the PTEs to indicate both (a) the ELF file where they are located and (b) the offset within the file where the content is located, or any information needed to perform the on-demand load. In many cases the page content may already be page-aligned within the ELF file, but if page aren't completely occupied this may not be the case. Then, as the process starts to run and generates page faults on regions of its address space that are mapped to areas of the ELF file, the OS loads content from the ELF file directly into newly-allocate pages. As an additional optimization, read-only code pages that are backed by ELF files never have to be swapped out, since they can always be reloaded from the binary. Note that this also enables page sharing between identical binaries, so every copy of `/bin/ls` isn't swapping in and out identical code pages. Nice.

Unfortunately, there is a wee problem with this approach, namely what happens if the binary is modified *while* the process is executing? This could create inconsistent content between parts of the process that have been paged into memory and parts that haven't, and havoc could result. Addressing this requires some kind of integration with the filesystem storing ELF binaries to prevent or at least trap accesses to binary files that are being used to back running processes. Preventing accesses obviously solves the problem, but has unfortunate consequences. Trapping accesses would allow the OS to load any pages from the affected binaries into memory to allow existing processes to continue running without problems.

Statistics:

- **8 out of 147** students answered this question. (Ballers!)
- **20** was the median score.
- **18.75** was the average score.
- **1.79** was the standard deviation of the scores.

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 shortest two. Complete these questions on the attached scratch paper. Clearly label your answers.

10. (25 points) OS Implications of Fast, Cheap, Non-Volatile Memory

Since the dawn of computing OS designs have been forced to make price, performance, and capacity tradeoffs when managing memory and secondary storage. Memory is fast but expensive (per byte) and volatile. Spinning disks are cheap (per byte) and non-volatile but slow. Flash is also non-volatile but much slower than memory and more expensive than spinning disks. To a large extent, these tradeoffs have driven the design of modern operating systems.

Now, imagine that you can cheaply provide a device with a terabyte of fast and byte-addressable (like memory) but non-volatile (like disk) storage. This isn't science fiction—the architecture community is exploring the potential of next-generation NVRAM chips that overcome the limitations of Flash. So let's do some dreaming...[4]

Present and motivate *five* different significant aspects of OS design that you would reconsider if you were designing an OS for a device with a single large and fast byte-addressable NVRAM chip replacing both memory and the disk (5 points each)[5].

Five may seem like a lot, but there are dozens of ways that this could revolutionize OS design. Think through the various subsystems that currently manage or use memory and the disk. Think about various OS operations that move state back and forth between memory and the disk. Think about how memory and disk are managed differently and how you could unify management of a single NVRAM chip. Think about process startup and shutdown, installation and update, state maintenance, and the effect of software bugs. Think about reboot. Consider big parts of the OS that may no longer need to exist, but also about side effects of the volatile nature of memory that you may want to preserve on NVRAM systems. Most of all: have fun!

[4] Thanks to Katelin Bailey, Luis Ceze, Steven D. Gribble and Henry M. Levy for inspiring this question.
[5] You can assume that we continue to use processor caches.

Graded by Jinghao Shi.

Rubric:

- **+5 points** per well-explained reconsidered design aspect.

Solution:

If you want the real complete solution to this question, look up this paper from the 13th Workshop on Hot Topics in Operating Systems (HotOS'11). The solution is essentially just paraphrasing their ideas.

Here's a set of eight, but there could be more.

1. **Goodbye paging and 4K page sizes.** Quite obviously, a device without both disk and memory no longer needs to move pages from disk to memory. Remember our goal when choosing pages to evict? We wanted to make the system look like it had as much memory as the disk size, all of it as fast as memory. Mission accomplished! This is an entire large OS subsystem that we can largely remove.

   In addition, remember that we chose a page size for a variety of reasons that had a lot to do with the backing store. When swapping, large pages minimize the size of kernel data structures while also amortizing the cost of disk seeks inside the swap file (or disk). But we are no longer swapping. Obviously, however, page size still plays a role in terms of hardware-aided address translation via the MMU and TLB, but because the NVRAM is byte-addressable we can experiment with new pages sizes more suitable given this radical architectural change. Some memory allocation and protection mechanisms even eliminate pages entirely.
2. **Unifying memory and filesystem protection.** Because we no longer have a separation between the memory and the disk, we need to consider how to unify the protection models. Memory protection is hardware-enforced on page boundaries. In contrast, filesystems provide more coarse-grained (file-level) but richer protection semantics, including the idea of ownership, group permissions, permission inheritance (via directories), and even the flexibility provided by capability-based access control lists (ACLs) on some filesystems. Part of this richness, obviously, is because filesystem protection mechanism are implemented in hardware, rather than software.

   But now with a single large NVRAM chip serving as both memory and disk, we might want to consider a way to unify these. At minimum, we have to ensure that the protection semantics provided by hardware when the NVRAM is used "like memory" are more closely-matched by what the OS provides in software when the NVRAM is used "like a disk".
3. **Unifying and memory and filesystem naming.** Another fundamental difference between memory and filesystems we need to reconsider is the idea of address spaces and how virtual addresses are translated. Address spaces essentially provide each process with a separate *local namespace*—recall that a virtual address

is meaningless without an address space to translate it inside, or alternatively without knowing what process is translating the virtual address. In contrast, filesystems provide a *global namespace*: `/path/to/foo` is the same file regardless of which process is opening or accessing the file. Unifying these two abstractions requires identifying and addressing this issue. One approach is to move to single address space machines, which have been previously explored.

4. **Installing, packaging, and launching apps.** Today, executing an app essentially transforms it from one form—on disk—to another—in memory. This is the job of the ELF file loader during `exec()`. A single NVRAM chip replacing both the disk and memory makes this unnecessary, and apps can be distributed in their ready-to-run form, eliminating the job of the ELF loader. Another big part of the OS gone! Although you still need to consider how to reconcile the addresses the process wants to use with the NVRAM that's available on the machine, but loadable libraries have approaches to doing this that could simplify the process.

   This change also fundamentally alters how application save state. Currently processes have to write out to stable storage to save state and have developed elaborate and process-specific mechanisms for doing so. A big chunk of non-volatile memory makes that unnecessary, and allows them to essentially be stopped and restarted in any state without any additional process support. So this really completely eliminates the usual process of "starting up" and "shutting down" a process as we currently know it, with no impact on memory consumption. Cool! Finally, NVRAM also makes it extremely easy to move process setups from machine to machine, stored exactly at any point in their execution.

5. **Application faults.** At last, a downside! One of the interesting side effects of NVRAM is that what used to be impermanent (RAM) is now permanent. Starting up and shutting down processes isn't done only to let the OS know that you are done using a particular program and allow its resources to be freed—it also creates a chance for the process to reload its memory contents from a known good starting point frozen in the ELF file. This is particularly important if the process is restarting to recover from a fault. So removing this capability entirely isn't necessarily a great idea, since whatever state corruption led to the fault is now impossible to remove.

   There are a few ways to address this without forcing processes back to storing state "on disk". One way is to have users create snapshots or checkpoints of the process at good states, which could be stored in the (very large) NVRAM and reverted to after failures.

6. **Decoupling power cycles and reboot.** Another interesting effect of the single large NVRAM chip is that power cycles no longer have to trigger a reboot. If the device loses power, (almost) all of its state is permanently stored on the NVRAM, with the exception being any register state associated with its execution at that precise instant. (Although with a small amount of battery backup that could

be written to the NVRAM after a sudden loss of power before the processor went offline.) So the idea that power cycles need to trigger a reboot on wall-powered devices no longer holds. On battery-powered devices the story is even simpler, since they already have ways to monitor their battery levels and don't need to be unprepared for most power outages—with the exception being the user suddenly ripping the battery out!

7. **Reboot and data corruption.** While this sounds like a plus, reboots for the OS can serve the same useful purpose as restarts do for processes: the chance to reload potentially-corrupted state from a known good starting point. So it probably makes sense to preserve some kind of reboot mechanism that can be triggered by users when needed that allows the OS to reinitialize state and recover from faults it may be unable to detect. But again, there is no need for reboots to be coupled in an way to power cycles, since the "memory" never loses state.
8. **New sleep states.** As previously mentioned, the overhead of entering a low-power sleep state is now limited to the cost of unloading the process registers into the NVRAM and then powering down the processor. On most mobile devices, the memory must either be moved to stable storage in order to be powered off—an expensive process—or kept in an active state to avoid losing contents, which consumes power. Our new device avoids this entirely and can sleep both much more quickly and completely and power on much more rapidly.
9. **Moving data between machines.** Finally, given the mixed nature of the content on the NVRAM, it may become more difficult to move data between machines. Part of this depends both on the protection and naming mechanisms adopted for a unified memory and disk storage device, as well as the ways that processes adapt to this change. For example, if processes choose to begin bypassing the file abstraction and store data that is usually store in files in memory—which is safe, since memory is permanent—it becomes impossible for users to move files from place to place. As a concrete example, if iTunes starts just storing all my MP3 content in memory, how do I reorganize (again, for the 12th time) my music collection? There are no files to manipulate! Or what if I want to move a picture to a thumb drive to show to a friend? But maybe filesystems and files are dead and over anyway, and all data transfer in the future will take place through the cloud. Who knows.

Statistics:

- **99 out of 147** students answered this question.
- **21** was the median score.
- **19.08** was the average score.
- **5.79** was the standard deviation of the scores.

11. (25 points) OS Energy Management

At the beginning of the semester we identified two main OS responsibilities: (1) providing abstractions and (2) multiplexing system resources. We then looked at three case studies of how this is accomplished for three core system resources: the CPU, memory, and the disk. In each case we discussed both how the OS performs and enforces resource allocations, and abstractions it provides to simplify resource usage.

A newer but equally-important resource for operating systems to manage is *energy*. Energy consumption matters at both ends of the computing spectrum, including for data centers for cost and power capping reasons, but particularly on mobile devices. For this question let's focus on smartphones—after all, consumers have been complaining for years about their short battery lifetimes.

First, consider how energy as a resource differs from other system resources we have discussed (10 points). These differences are significant from the perspective of designing effective approaches to managing energy. You might want to use memory management as a point of comparison, since several of the OS requirements for multiplexing memory (grant, enforce, . . .) don't really have appropriate analogs with energy, and several new capabilities are needed.

Second, present a detailed design allowing operating systems to manage process energy consumption on battery-powered smartphones, describing any changes needed at the hardware, OS, and system call interface layers (15 points). Be careful not to make assumptions about OS capabilities regarding energy. Compared with the resources we have discussed, managing energy has some unique prerequisites that complicate the problem. It might be helpful to consider, as a starting point, an OS that knows *nothing* about the energy consumed by processes and go from there. You don't need to provide abstractions unless they are required, but your solution should enable multiplexing. You may want to consider typical usage of smartphones as well as charging patterns as part of your design.

Graded by Jinghao Shi.

Rubric:

- **+10 points** for identifying differences between energy and other system resources
- **+15 points** for a design allowing operating systems to manage energy

Solution:

There were two parts of this question, clearly identified within the question. The first asked you to identify differences between energy and other resources that would be significant for energy management. The second asked you to design a way for the OS to manage energy.

First, there are several important differences. One is that, unlike other system resources, energy is *depletable*. Once it's consumed, it can't be reused, nor can previous allocations be revoked. And when it's gone, the device cannot be used. Contrast this with the processor, which the device never "runs out of": as long as the device is on, there is more processor time. For the disk and memory, the device can run out, but at that point both memory and storage can be reused by revoking previous allocations. Memory can be reused by swapping out pages or, in the worst case, simply by killing processes, in both casing creating new unused pages for a new process. Disk space can be reused by prompting the user to remove old files.

As a result, bad OS energy management has the potential to be more damaging than poor management of other resources. Consider what happens if the OS makes a bad scheduling decision: the wrong thread or process runs for a few tens of milliseconds, and then the OS has the chance to make a better scheduling decision. The device isn't going to have to power down as a result. Also with memory—if the OS swaps out the wrong page, it may have to swap it right back in, hurting performance. But again: the device isn't going to power off, and the contents of the memory page are preserved. But make a poor energy allocation decision and you may have just reduced the device's lifetime by seconds and can never recover from the mistake. So clearly this is important!

A second significant difference is that energy is consumed implicitly by the usage of other system resources, not directly. Processes don't consume energy in order to consume energy, the consume energy in order to use the processor, or read and write to memory, or send data over the network interface, or draw to the display. Pretty much *every* system component consumes energy to run, but the point isn't to consume energy—it's to accomplish something else. Energy consumption is a side effect.

Second, as a guide to our design let's return to our memory multiplexing requirements as the question suggested and see if we can modify them for energy management. They were:

- **Grant:** the kernel should be able to allocate the resource to processes.

- **Enforce:** the kernel should be able to enforce resource allocations efficiently.
- **Reclaim:** the kernel should be able to reuse the resource if it is unused.
- **Revoke:** the kernel should be able to stop a process from using resources it was previously allocated.

Clearly #4 doesn't apply, since once energy is consumed it is gone. However, we should be able to design mechanisms to accomplish the other three. In addition, energy consumption presents a new requirement: **measurement**. We had really been relying on this all along: in order to enforce allocations, the kernel has to be able to measure the resource being used. But with energy it's complex enough to deserve its own bullet point. (We'll get back to that in a minute.) So what we have is:

- **Measure:** the kernel should be able to measure the amount of energy used by each process.
- **Grant:** the kernel should be able to allocate energy to processes.
- **Enforce:** the kernel should be able to enforce energy allocations efficiently and prevent processes from using more energy than they were allocated.
- **Revoke:** the kernel should be able to stop a process from using energy it was previous allocated but has not yet consumed.

Let's go step by step.

Measure: this isn't as trivial as it sounds. In fact, it's about half of the battle when managing energy. There are two main challenges.

First, each hardware component consumes energy differently: one CPU consumes more energy than another, and may consume a different amount of energy depending on which frequency level it is running at. Using a Wifi wireless network consumes less energy-per-byte than mobile data networks such as 3G or 4G, and more than using a wired network. Spinning disks may consume more energy per operation than Flash.

Normally systems work around this by using a *power model*, which allows energy consumption to be estimated in software based on usage of the component. For example, if a process sent a certain number of bytes over a particular network interface, then it consumed a certain amount of energy to do so. Power models can be quite complicate, however—when estimating network energy consumption you might need to incorporate things such as the signal strength of the network connection at the time that the data was sent. In addition, each component requires its own power model, further complicating the OS.

Alternatively, we could abandon the power model and simply rely on each hardware component to perform its own energy measurement, exposing the energy consumed to the OS through a hardware interface. This is a nice way of avoiding the problem while still respecting hardware differences.

Second, it can be difficult even to figure out track *what* process is using which resource. When resources are used synchronously—i.e., you have to be running on the CPU to use them—then this is somewhat simplified. (But what about memory energy consumed on a multi-core system?) But asynchronous resources pose their own problems, partly due to layering caused by good interface design. For example, by the time a disk write reaches the disk driver—which knows the power model—it may no longer identify the process that issued it, which also will not be the process currently running. Solving this problem is an open engineering problem, so we didn't expect you to—but you would receive credit for identifying it.

Grant and Enforce: if you can measure, you can grant and enforce. Granting energy simply allows the process to consume it. Depending on your design, you might have exposed energy grants to the process in some way, or not. Telling the process how much energy it is allowed to use may allow it to make better decisions about how to execute. Alternatively, it may have no idea how to use this information. So there are arguments both ways.

Enforcement, however, is a bit different due to the implicit nature of energy consumption. If the OS wants a process to use less memory, it can move its pages to disk. If it wants a process to use less CPU, it can schedule it more often. However, if it wants a process to use less energy, it has to prevent it from using other system resources—any of them—that would consume energy. The simplest and most effective thing to do is simply stop it from running. That way it can't get the the processor to run, use memory, allocate memory, or use the network or other peripherals, all of which would consume energy.

Revoke: this one is a bit interesting in the case of energy. To allow processes to plan around their energy allocation, you might want to give them a bundle of energy to use and then force them to request more when need it. Various research operating systems have explored ideas like this, using abstractions like energy "tickets" which express a reservation on some of the energy remaining in the battery.

The problem is hoarding: what does the OS do if a process requests energy that it doesn't use? This could happen because it doesn't understand its own energy consumption, which is possible, or simply because it wasn't run as often as it thought it would be. In either case a revocation mechanism might be required to allow the OS to reallocate available energy by invalidating previous unused allocations.

Allocation Policy: a final issue that you should have addressed is *how* to allocate energy, a unique challenge given its depletable nature. Resources such as the CPU and memory are usually allocated as needed based on some prioritization scheme, but allocating energy this way doesn't really allow the OS to perform energy management—it just ends up being purely a side-effect of other allocations.

Instead, it might be worth considering a goal of energy allocation, similar to how we discussed interactivity or throughput as potential goals of processor scheduling.

One possible goal that many previous systems have explored is the idea of meeting a lifetime target. Given a length of time, say 8 hours, the OS is in charge of metering energy consumption to ensure that the device lasts for at least 8 hours. Users could configure this based on their charging patterns to ensure that the device would never run out of energy before reaching a plug. One way to accomplish this is to simply divide available energy over that time interval and meter it out evenly. At any point in time, if the device is over its energy budget processes must stop running; if it is below the budget, they can run freely.

Unfortunately, this isn't necessarily a great idea. First, smartphones don't consume energy smoothly but in bursts. I may use my smartphone intensively for an hour and then not again for four hours. If I meter smoothly, then I'm bumping up against the energy budget repeatedly during the first hour and reducing performance, but then have built up a big surplus during the interactive period. Another problem is that stopping interactive tasks that have run out of energy is bound to frustrate users. Instead of taking 10 s to load, my energy-limited browser takes 1 min to load the same page because of an energy budget. Who's waiting? Me.

We don't really have great solutions for these problem yet. If you do, come talk to us!

Note that this question was released to the class before the exam.

Statistics:

- **137 out of 147** students answered this question.
- **20** was the median score.
- **18.73** was the average score.
- **5.72** was the standard deviation of the scores.

12. (25 points) Smartphones, Meet Cloud

Today's computing ecosystem is increasingly dominated by the interaction between two "devices"—smartphones and the cloud—with complementary characteristics. The cloud is always-on, wall-powered, has large amounts of computational power as well as memory and storage capacity, but is located at a datacenter far, far away[6]. In contrast, smartphones are also always-on and always nearby, but have capabilities constrained by multiple factors: energy, form, price, and heat dissipation. It seems likely that smartphones will never be as powerful as the cloud, but neither will the cloud ever be as nearby as your smartphone. It is also much easier to make the cloud faster—just add more machines—whereas making the smartphone faster can be challenging—how do I get eight cores into the same spot where four used to fit, and without lighting a fire in someone's pants?

But what we want to achieve is for our smartphones to serve as a front-end for the cloud, making our smartphone *seem* as powerful as the cloud by intelligently arranging interactions between the two devices. First, make an argument using Amdahl's Law about how future improvements to cloud performance might be lost if cloud resources aren't used intelligently (5 points).

Second, apply two of the systems design principles that we discussed in class to improving the interaction between energy-constrained smartphones and the cloud (10 points each). You can make the following simplifying assumptions:

- The cloud and the smartphone are always connected.
- However, access to local smartphone resources—CPU, memory, and storage—is always much faster than accessing cloud resources due to network delays and limitations[7].
- The cloud can compute much faster and has much more storage available than the smartphone.
- Smartphones are energy-constrained but also regularly recharged.

For each, state the design principle (2 points) and why it is applicable (2 points), and then present a reasonable design sketch of how you propose to apply the principle to smartphone-cloud interactions. Each of your improvements should probably have the effect of either (a) improving user-visible performance, (b) reducing smartphone energy consumption or (c) providing smartphones access to some new capability that requires the cloud's resources.

[6] Not necessarily that far, but farther than your pocket.

[7] At some point the speed of light also starts to play a role.

Graded by Jinghao Shi.

Rubric:

- **+5 points** for making an argument using Amdahl's Law about the relationship between cloud and smartphone performance
- **+10 points** for each systems design principle applied to the problem

Solution:

There were two parts to this question. First, we asked you to make an argument using Amdahl's Law about why future improvements to cloud performance might be lost if interaction between the cloud and user-facing devices such as smartphones isn't managed intelligently. Second, we asked to apply two systems design principles and describe how they might apply to the interaction between the powerful cloud and energy-constrained smartphones.

So first, the Amdahl's Law part. This was intended to be easy and reminiscent of the argument made by Patterson et. al when motivating RAID. If some portion of my performance is determined by the cloud, and the remainder is determined by the end-user smartphone, then Amdahl's Law states that the overall system performance is constrained by part that's not improving. So as the cloud grows faster, its contribution to overall performance decreases until performance—and conversely, slowdowns—are entirely determined by the smartphone. Put another way, even if the cloud could perform its part of the problem *instantaneously*, we'd still be left with the parts that the cloud isn't responsible for that take place on the user-facing device—as well as the latency to move data back-and-forth between the device and the cloud.

Now, let's apply some of our favorite systems design principles to the problem of cloud-smartphone interaction. In no particular order:

1. **Add a cache**, or as Lampson puts it: cache answers. Given that the cloud is both powerful but also far away, it would make sense to put a less powerful (or smaller) device between the user and the cloud. On some level that device already *is* the smartphone, so we already have a cache. However, you might have identified other devices that could be used as caches that are closer to the smartphone than the cloud. How about other personal devices located on the same local network, like desktops or laptops? Desktops aren't energy-constrained and are likely to be quite powerful and have a great deal of storage. And although laptops are energy-constrained, they are still more powerful than the smartphone. So these kind of devices are options.

   However, just identifying that the smartphone or other devices can serve as a cache doesn't answer some fairly important questions about this technique. First, caching *what*? Second, how do you manage the cache to try and hide latency associated with contacting the cloud? One example would be effectively treating the smartphone's local storage as a cache for a filesystem where all the

users file were stored in the cloud. So users would have access to a potentially-unlimited (or at least very large) amount of content, much more than could be stored on the 16 or 32 GB of Flash storage located on their smartphone. This storage would also be more reliable, since it could be replicated in the cloud across multiple machines. So losing your smartphone would not mean losing access to all of those priceless selfies you took recently! Still, this cache would have to be intelligently managed, given that there would be a large latency difference between opening a local file and opening a file stored on the cloud. You could apply some of the same algorithms here that we applied to page replacement.

2. **Use the past to predict the future**, or as Lampson puts it: use hints. Speaking of intelligent cache management, it could really improve device-cloud interaction if we could anticipate what the user was about to do so that we could retrieve data from the cloud *before* it is needed and avoid the latency. So before they start up their photo-browsing app to review all of their recent selfies, the smartphone could pull all of those photos down from the cloud and store them locally. And right as the user opens their email client, all of their email would have just synced and new messages waiting and available. An even more powerful application of this would be to help predict charging sessions, since this both helps in managing available energy and in determining whether certain delay-tolerance tasks—such as updating apps—can be delayed until the device reaches a plug.

   Obviously we don't have a crystal ball here, but we can always apply our old trick of using the past to predict the future. What might help drive our prediction engine? For charging sessions, we could build up a model of what times of the day the user normally charges, or how long their typical discharging sessions last. For app activity, we could determine what the probability is that users launch certain apps based on the time of day, or other apps that they have just opened, or even based on their location. Smartphones have the benefit of offering a great deal of additional context information, such as location, that might work well as inputs into these prediction models.

3. **Procrastinate.** Related to using the past to predict the future is the idea of procrastination. We've talked about cases where procrastination allows the OS to avoid doing things entirely, such as loading unused code pages by performing demand paging. In the case of smartphone-cloud interaction, procrastination might serve that purpose, but it's more likely that what we want to do is avoid doing things *when discharging* that we can wait to do until we are charging, particularly things that consume a great deal of energy. As mentioned previously, bulk data transfer between the device and the cloud is an example. Imagine that a user orders a movie from the cloud to be delivered to their smartphone, but can set a deadline by which the movie should be on their smartphone and ready to view—say the point at which they leave their office for the train ride home. Now, how do the cloud and the smartphone arrange the transfer so that it both (a) arrives by the deadline and (b) consumes as little energy as possible, by

avoiding the high energy overhead of transfers over mobile data or poor-quality Wifi networks. Obviously this involves a bit of procrastination, since at times the smartphone wants to wait until a better network is available. In addition, if the user ever plugs in their phone during the day, then the transfer can be performed without consuming any energy at all—this is the best case.

4. **Compute in the background.** This design principle we have to twist a bit, but it's quite apropos. On a single device the idea of computing in the background is to try and do work while the device is idle so that we can avoid doing things when it is busy. Page cleaning—synchronizing the contents of memory with the contents of the swap file—is a great example of this, since it moves swapping disk activity into idle periods and off of the swap out path. One way to apply this to smartphones is just to try and do things while on plug—we've discuss this previously. However, a better idea is to exploit the fact that we have the powerful cloud nearby! So instead of "compute in the background", we can adjust this design principle to be "compute in the cloud".

   Here's an example. A user takes a photo and the photo recognition software wants to analyze it to automatically determine what friends are pictured so that they can be automatically tagged. This is potentially a computationally-intensive activity, and performing the face detection on the smartphone will consume energy. But instead, the entire process can be *offloaded* to the cloud. The cloud is sent the picture, runs the face detection algorithm, and returns a list of friends identified in the picture and their face locations.

   Sounds like a great idea! But there's several catches: moving data to the cloud takes both time and energy. Let's think about time first. If the time to perform the transfer is much longer than the time to run the algorithm, it may hurt performance to offload the computation. Think of Shazaam as an example: audio samples are fairly large, and so the signature detection is performed on the mobile device while the library mapping is performed in the cloud. Uploading the entire audio clip would make the whole process take much longer. So there is a balance between computation and data transfer speeds to consider.

   Second, let's think of the energy issue. Remember that our goal here was to either improve performance or save energy by offloading to the cloud—preferably both! But in some cases cloud offloading doesn't actually save energy either, and can actually consume more. The argument is similar to the one about performance that we've just gone through, but instead of performance we have to consider the balance between energy for the transfer and the energy required to compute locally, since not involving the cloud doesn't require the we transfer anything to the cloud. As an additional wrinkle, data transfer energy consumption fluctuates, so cloud offloading may make sense when on an energy-efficient Wifi network but not while on an energy-hungry mobile data network.

Statistics:

- **56 out of 147** students answered this question.
- **20** was the median score.
- **18.68** was the average score.
- **6.82** was the standard deviation of the scores.