CSE 421/521 Final Exam Solutions

—SOLUTION SET—

15 May 2017

This final exam consists of four types of questions:

1. **Ten multiple choice** questions worth one point each. These are drawn directly from the 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 mostly (but not entirely) drawn from second-half material.
3. **One medium answer** question worth 20 points drawn from second-half material. 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. **Answer *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.**

Multiple Choice

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

   (a) Which university should GWA work at next?
   √ **Illinois** √ **Northeastern** √ **Duke** √ **Stony Brook**

   (b) Significant differences between file systems include everything *except*
   ◯ on-disk layout. √ **reliably storing Vicky's data.** ◯ data structures.
   ◯ crash recovery mechanisms.

   (c) What is a hint that a page might be good to swap out?
   √ **It hasn't been used for a while** ◯ It's currently loaded into a core's TLB
   ◯ Ali doesn't like it ◯ It's shared by multiple processes

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

   (e) 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.

   (f) 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.**

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

   (h) It is the *most* difficult to get repeatable performance results when
   ◯ asking Carl. ◯ running a system simulator. √ **measuring a real system.** ◯ using a system model.

   (i) Applications will run exactly the same in a virtual machine as they would on real hardware.
   ◯ True √ **False**

   (j) Which of the following makes it *easier* to virtualize the x86 architecture?
   ◯ hardware page tables ◯ instructions that are not classically virtualizable
   √ **multiple privilege levels** ◯ Guru

Short Answer

Choose **4 of the following 6** questions to answer. You may choose to answer additional questions, in which case you will receive credit for your best four answers.

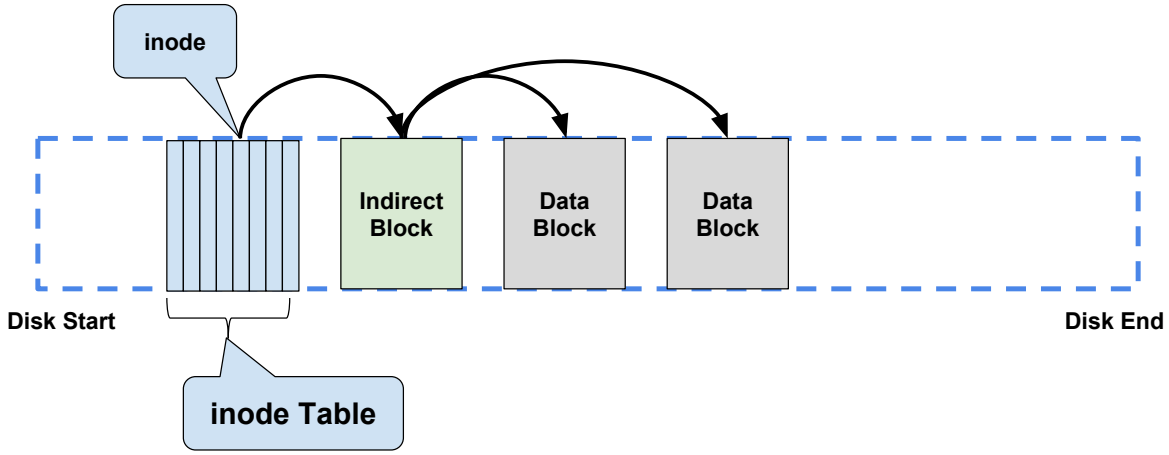
2. (5 points) Name five different "names" that the operating system must virtualize to provide container virtualization.

Solution:

1. Process IDs
2. User names
3. Group names
4. Device names
5. File names and mount points
6. Host names
7. Host ports

3. (5 points) Draw a diagram of a traditional (or `ext4`-like) file system layout. Label the `inode` table (2 points) and show an example file, including `inode` (1 point), indirect block (1 point) and data blocks (1 point).

Solution: Here is an example:



4. (5 points) Log-structured file systems rely on several assumptions to perform well. Identify one of those assumptions (2 points) and describe a realistic workload that would violate it (3 points).

Solution: Two of the more important assumptions that we presented were (1) that a large cache will soak up most reads and (2) that cleaning can be performed without interfering with ongoing disk activity. Either one of these being violated will result in seeks to other parts of the disk, and limit our ability to leave the disk head in the same place for long periods of write activity.

An example workload that violates (1) would be one where the pattern of reads does not have enough locality to fit inside the cache. Imagine a rapid linear scan through an array of disk blocks that is twice the cache size, or large. By the time the reads for the second half of the array have completed, the entire first half has been evicted and has to be read in again.

An example workload that violates (2) is a long-running workload with no breaks in disk activity, and particularly one that creates a lot of holes in the log. Eventually LFS will need to clean, and because disk activity is ongoing that cleaning will interfere with normal reads and writes to the active log.

5. (5 points) Describe the big idea behind Redundant Arrays of Independent Disks (RAID) (3 points). Identify another manifestation of that big idea in a different computer system (2 points).

Solution: As stated on the lecture slides, the big idea is that several (or many) cheap things can be better than one expensive thing.

Other manifestations of that idea are everywhere. Cloud services use many servers to create one hugely powerful supercomputer, and must deal with the fact that individual machines are failing constantly. Many crowdsensing techniques use many unreliable but cheap human or low-quality sensor inputs to try and replace more expensive but more accurate sensors. Even multicore chips are on some level a manifestation of this idea, since as single-core CPUs became harder to cool they were replaced with multiple simpler cores.

Note that this question was released prior to the final exam.

6. (5 points) Explain the core cost-benefit tradeoff faced when swapping pages to disk. What is the cost (2 points)? What is the benefit, and how can it differ (2 points)? What is a clever way to reduce the swap-time cost to zero (1 point)?

Solution: The cost is the I/O required to transfer 4K (or whatever the page size is) to and maybe from the disk. The cost is usually fixed, although if you can choose a page to evict that will never be used again you can drop it to half of the normal case.

The benefit is use of the memory that that page consumed for as long as it remains unused. As a result, the benefit depends on the amount of time that the page remains on disk. The longer it is swapped out, the longer the memory is available and the large the benefit is.

Background page cleaning to synchronize in-memory page contents with on-disk contents can remove the need to write the contents to disk when the memory is needed, and thus reduce the swap-time cost to zero—or almost zero, just the time required to update the page-table entry, which is zero-ish.

Note that this question appeared on the 2016 final exam.

7. (5 points) Define Amdahl's Law and describe how it guides the process of performance improvement.

Solution: We discussed two different formulations of Amdahl's Law:

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

—or—

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

We also had our corollary to Amdahl's Law:

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

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

Note that this question appeared on the 2015 final exam.

Medium Answer

8. (20 points) Virtualization Comparison

We discussed three types of virtualization in class: full hardware virtualization, paravirtualization, and OS or container virtualization. First, clearly describe each type of virtualization and give a brief overview of how it works (5 points each). You should explain what is virtualized, define the pieces of software that are involved, explain any constraints that this virtualization places on the virtualized environment, and identify any key challenges to this virtualization approach.

Second, list three virtualization use cases that motivate each of the three approaches (2 points each, up to 5 points total). For each of your examples you should make a convincing case that the other virtualization approaches are impossible, ineffective, or perform poorly.

Solution:

1. **Full hardware virtualization.** What is virtualized is the hardware interface. The VM environment is provided by the virtual machine monitor or VMM. Full hardware virtualization allows an *unmodified* guest OS and applications to run safely inside the VM. To allow this to be done safely the guest OS is run without full kernel privilege but instead with a privilege level typical to user applications. When the guest OS tries to do something that requires kernel privilege, ideally a trap will occur which the host OS will pass to the VMM to handle. Unfortunately, on the x86 architecture many instructions do not trap properly and must be dynamically rewritten at runtime to be handled safely. Full hardware virtualization is required any time that the guest OS cannot be modified, but has worse performance than other approaches due to the complicated path that traps and exceptions must take to return to the VMM.
2. **Paravirtualization.** What is virtualized is the hardware interface. The VM environment is provided by the virtual machine monitor, but it is typically referred to as the hypervisor in this case. Paravirtualization requires small changes to the guest OS to replace privileged instructions with calls to a new API provided by the hypervisor, which inspects and handles all privileged operations that could pierce the VM. For paravirtualization to perform well, interaction between the guest OS and the hypervisor must be efficient. Paravirtualization is an excellent choice when OS-level isolation is required, or in cases where the virtualized hardware must support multiple operating systems. It generally outperforms full hardware virtualization which should not be used if paravirtualization is an option.
3. **OS virtualization.** What is virtualized is the OS namespace, allowing full isolation between multiple sets of applications running on top of the same OS, each in their own container. In this case the OS itself provides the virtualized environment by

extending names with container identifiers which allow it to distinguish between the resources used by processes running in different containers. OS virtualization requires considerable support by the OS itself, which has to make changes to every part of the OS that uses a variety of different names: file names, process IDs, network identifiers, etc. OS virtualization also requires that all containers use the same underlying OS and version of the OS interface. OS virtualization is effective at packaging and distributing integrated software packages because containers can contain multiple apps and isolate their configurations from other software packages running on the same system—even other version of the same software running in the container.

Note that this question was released prior to the final exam and appeared on the 2016 final exam.

Long Answer

9. (25 points) Learning By Doing

A significant part of this class is completing the large programming assignments: `ASST2` and `ASST3`. Most of you will probably not work on these assignments again—although Carl has apparently implemented signals and completed ASST4. But hopefully you have learned some lessons from your struggles with OS/161 that you can apply to your future projects.

To answer this question, discuss **five** things that you learned by completing `ASST2` or `ASST3`. (If you did not complete or attempt either of these assignments, please answer the other two long answer questions.) In each case, describe what you learned and how you learned it (3 points) and how you will apply this lesson to your future projects (2 points). You can talk about tools, design, working with your partner, time management—whatever you want, as long as it is something that you learned this semester from the OS/161 assignments.

Rubric:

As embedded in the question, for each lesson:

- **+3 points** per lesson what and how, and
- **+2 points** per future application.

Solution:

Answers to this question are fairly personal. But they could include things like the importance of design, proper coding style, use of Git, how to ask questions effectively on the forum, effective ways to divide up large amounts of work, the utility of unit test cases, and other such useful life lessons.

10. (25 points) Unikernels

Traditionally operating systems were designed to support multiple applications running together on the same machine. This was a reflection of an era when most machines—both desktops and servers—typically hosted many different users and applications.

Today, virtualization allows us to easily set up an entire machine dedicated to running a *single application*. Operating systems are beginning to evolve to support this use case. So-called **unikernels** are operating systems specifically tailored to supporting a single application: for example, a web or database server, or scientific application. The single application can run as normal using whatever operating system constructs it would normally use. However, the operating system below it has shed many of its traditional features as it only needs to support that single application. This question explores some of the implications and opportunities of the unikernel design.

First, being able to customize an operating system for a specific application places requirements on the design of the operating system itself. Identify and discuss these requirements (10 points). As a (big) hint, unikernels are usually built starting with so-called *library* operating systems.

Second, list three beneficial ways that the underlying operating system can be customized to support a single application (5 points each). For each, describe what is beneficial about that particular customization. To receive full credit, please include at least two different *kinds* of customizations.

Rubric:

As embedded in the question.

Solution:

This question was inspired by recent work on unikernels, which you can read about here.

For the first part of the question we would accept (at least) two answers. First and foremost, unikernels require that the operating system itself be able to be refactored to meet only the needs of the single application it has been customized to support. For example, if that application doesn't need a file system, then the file system portions of the unikernel need to be able to be removed. If that application doesn't need a network stack, then that network stack needs to be able to be removed. One of the benefits of unikernel is that, by shrinking the amount of code that constitutes the operating system, what results is simpler and shrinks the attack space—making the overall system more secure.

Traditional operating system designs have tended to become what is called *monolithic*. That means that they merge together and create dependencies between lots of different parts of the system. In many cases this can improve performance, but in others it is just the result of sloppy design. Regardless, these dependencies make it difficult to

remove one large portion of the system, a feature that unikernels require. For example, if you completed ASST3.3 you might think about how hard it would be to now remove swapping from your kernel—or make it possible to both swap or not swap, depending on whether the unikernel was configured to use a disk or not.

This is why, as the hint mentioned, they are frequently built on so-called *library operating systems*. Library operating systems try to take a more modular approach to provide operating system functionality by breaking required pieces into different libraries. When creating a unikernel around the needs of a single application, the designer can pick and choose the libraries that that application needs. That has the affect of removing unnecessary code from the operating system, simplifying it, and making the overall complete unikernel environment more secure.

A second answer that we would accept for the first part has to do with customization. Traditional operating system designs may end up adopting certain design choices or system parameters that work well for all applications. For example, we discussed earlier in the semester the fact that the Linux scheduler is "baked in" to the system and not modular. One opportunity when creating a unikernel is to tightly optimize the operating system for the single application it is supporting. So the designer must be able to perform deep customization of the operating system being used to ensure that all "one size fits all" choices are eliminated.

Here are some examples for the second part of the question drawn from three different categories. But we would accept other reasonable answers as well.

1. **Remove unnecessary components.** This was already discussed above. Parts of the operating system that are not needed by the single application can be removed. Because the application doesn't use these components, it will not miss them—and the unikernel has no need to support any other applications. We would accept a variety of different answers here with appropriate justification.
2. **Deep OS customization.** The application may want to choose its own scheduler (if it has multiple threads); customize paging, page size, or other caching behavior; optimize the network stack to its own behavior, etc. Again, there are a wide variety of different answers here that we would accept.
3. **Elimination of traditional protection boundaries.** One interesting opportunity created by unikernel operating systems is the ability to remove certain protection boundaries typically used to isolate applications from each other. For example, unikernels must only provide single address spaces given that they are only supporting one application. If you assume that the application is well-behaved, you may also eliminate some of the protections usually provided to prevent applications from crashing the operating system. If a unikernel application crashes, then essentially the whole system crashes. So if it does something that would violate a normal protection barrier and we let it crash the operating system, then that is a

pretty much equivalent outcome. That being the case, do we really need to provide memory protection for the OS, users and groups for the file system, etc.?

Note that these protection reductions have to be applied *by the operating system*, not by applications, given that the assumption is that the single application behaves identically in the unikernel environment.

As indicated in the question, it is important that your answer touch on several of these opportunities. We would not give full credit, for example, to a solution that just talked about removing or customizing three different parts of the operating system. As long as you touched on two of the opportunities above—or others that we have omitted—you will receive full credit.

11. (25 points) Asynchronous System Calls

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

First, considering the system calls we have discussed throughout the semester, describe several cases in which asynchronous system calls would be useful and how (3 points). Conversely, describe several synchronous system calls that lack a meaningful asynchronous analog (2 points).

Second, describe any changes to the operating system interface that might be necessary to support certain asynchronous system calls (3 points). Discuss any additional application programming challenges that non-blocking system calls may introduce (2 points).

Third, briefly explain how a process that can fork multiple threads can emulate asynchronous system calls without true non-blocking support from the operating system (3 points). Describe the overheads to this approach that might make native asynchronous system calls preferable (2 points).

Finally, describe the kernel changes necessary to support asynchronous system calls. Walk through the steps required to complete a non-blocking call, describing what happens at both the process (5 points) and kernel (5 points) level.

Rubric:

As embedded in the question.

Solution:

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

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

   Several system calls, however, really don't make much sense asynchronously. What does it mean to do an asynchronous `exec()` or `fork()`? In the former case, the process just returns and can run for a bit longer before it is replaced by a new image. This doesn't seem useful. With `fork()` you have the problem that the parent

wants to create a copy at a well-defined moment, and allowing it to return and continue running would complicate that process. (You could do this though, if you were careful, but again the use case doesn't seem obvious.)

2. Let's focus on the I/O related asynchronous calls, particularly `read()` and `write()`. What would we need to add in order to allow the process to return immediately? The first thing is that we need to provide a way for the process to determine *when the call has completed*. One way is to provide some additional system call that a process can use to poll for a pending I/O request; another way is for the operating system to send the process a signal when the requested I/O completes.

   The reason that this additional signaling mechanism is required is that the process must not modify or interpret the `write()` or `read()` buffer until the non-blocking call completes. This requirement also creates a new concurrent programming challenge which the application designer must deal with. With a blocking system call, there is no way for the process to see a `read()` in an incomplete state; with a non-blocking call it can, and care must be taken to ensure that it does not assuming the correctness of the program depends on it. (It probably does; imagine a web-server that served pages that had not been completely read from disk.)

3. If a process can create threads visible to the kernel it can emulate asynchronous calls. When a thread wants to perform a blocking system call, it forks a new thread specifically for this task. That second thread blocks across the system call while the original thread continues to run. When the call completes, the thread that was forked to perform the call exits.

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

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