CSE 421/521 Final Exam Solutions

—SOLUTION SET—

11 May 2015

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. **Two long answer** questions worth 25 points each, integrating material from the entire semester. 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) Our unexpected April 1st visitor was

   [✓ **handing out beads.**] [✓ **wearing a blue body suit.**] [✓ **not shy.**] [✓ **Nick DiRienzo.**]

   (b) What was one approach that Wickizer et. al used to improve Linux scalability to many cores?

   ◯ Increasing the length of OS critical sections [✓ **Sloppy counters**] ◯ Reducing application performance ◯ Asking Zihe

   (c) The RAID design principle can be best summarized as

   ◯ building expensive things from many inexpensive parts. [✓ **building reliable things from many unreliable parts.**] ◯ five levels. ◯ striping.

   (d) Filesystems use ________ to map filename components to inode numbers.

   ◯ inodes [✓ **directories**] ◯ superblocks ◯ Guru

   (e) After a crash, which of the following would indicate data loss for a journaling filesystem?

   ◯ multiple checkpoints ◯ journal entries after the last checkpoint [✓ **incomplete journal entries**]

   (f) Which of the following makes it *easier* to virtualize the x86 architecture?

   ◯ hardware page tables ◯ instructions that are not classically virtualizable [✓ **multiple privilege levels**] ◯ Jinghao's blog

   (g) Which of the following is *not* a good hint that a page might be a good page to swap out?

   ◯ It hasn't been used for a while ◯ It's clean [✓ **It's currently loaded into a core's TLB**]

   (h) Applications will run exactly the same in a virtual machine as they would on real hardware.

   ◯ True [✓ **False**]

   (i) Which of the following applications has become very similar to the traditional operating system?

   [✓ **The web browser**] ◯ iTunes ◯ Microsoft Word ◯ `sys161`

   (j) We discussed all of the following kernel designs *except*

   ◯ microkernels. ◯ monolithic kernels. ◯ exokernels. [✓ **xenokernels.**]

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) We have described (at least) two places where the data structure used by the OS presents a tradeoff between desirable properties. Identify one (1 point), describe what is is for (2 points) and describe the tradeoff (2 points).

Solution:

Here are the two places we had in mind, although there are probably others:

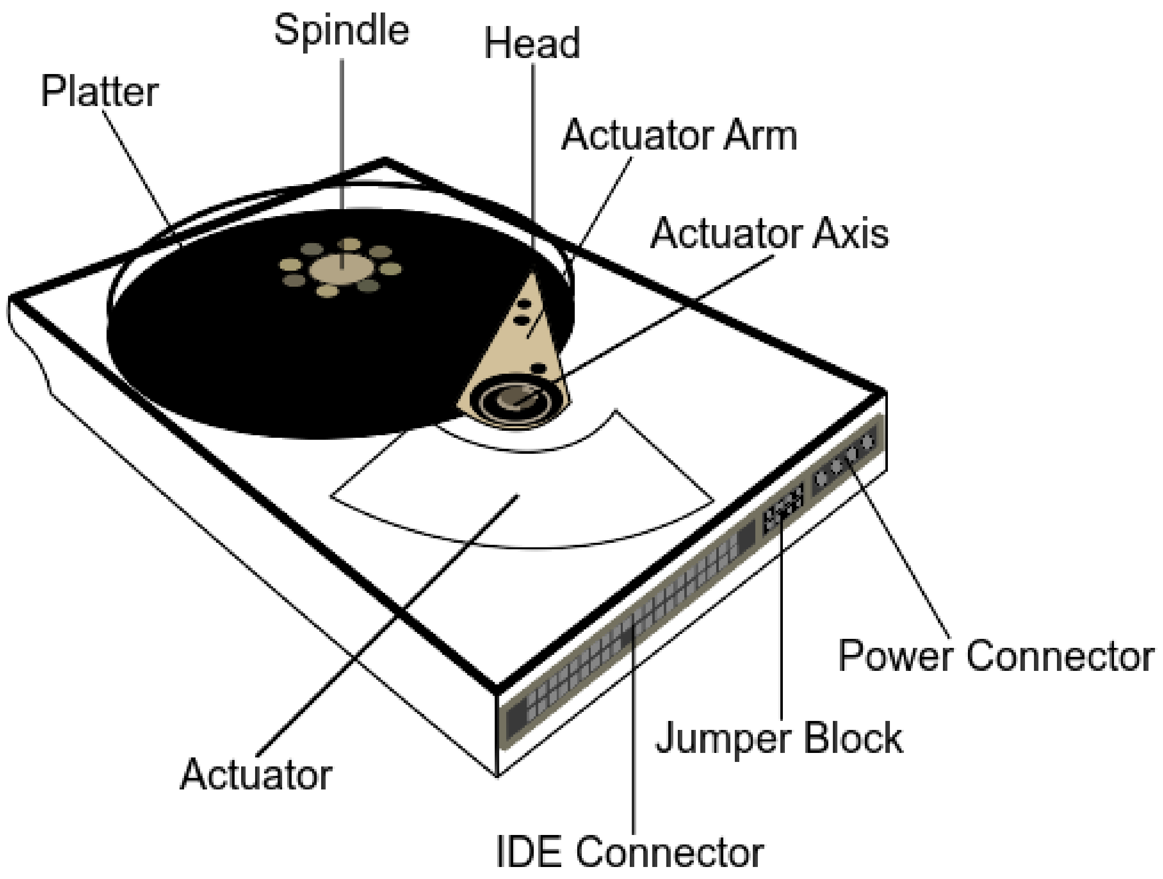
1. **Page tables.** Process page tables map process virtual addresses to page table entries (PTEs), which may contain the physical address (if the page is in memory), a disk location (if it is swapped out), or other information. Page tables present a size versus mapping speed tradeoff. Compact page tables (such as a linked list of PTEs) make mapping speed scale poorly as the size of the address space grows, while the O(1) page table (an array) is way too large. Multi-level page tables or lists of segments with their own lists of PTEs are both compromises between space and speed.
2. **Inode to data block mapping.** (We didn't really have a better name for this.) Allows filesystems to locate the ordered list of data blocks associated with a file starting from the inode structure. Presents pretty much the same size v. speed tradeoff as page tables, with the additional considerations that (1) files should be able to grow to arbitrary sizes (as opposed to fixed-size address spaces) and (2) traversing large on-disk data structures may involve multiple read operations that require seeks and are, therefore, slow on spinning disks. The use of (double, triple, quadruple) indirect blocks was another compromise that both allows files to grow to extremely large sizes while reflecting the fact that many files are small.

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

3. (5 points) Draw a simple diagram of a spinning disk. Label three parts (1 point each) and point out the major source of latency (2 points).

Solution:

Here is a diagram of a disk, although yours was (hopefully) probably a lot simpler:



You also needed to point out that the lateral movement of the arm to reposition the heads (seeking) is the major source of latency.

4. (5 points) Describe the difference between full virtualization and paravirtualization (2 points), and how one way that the guest OS manipulates the virtual machine is handled differently by the two approaches (3 points).

Solution:

Full virtualization attempts to present a virtual machine abstraction that is identical to the physical machine, which allows running *unmodified* guest operating systems. **Paravirtualization** presents a modified virtual machine abstractions that is easier for the hypervisor to support, but *requires changes* to the guest OS.

One example difference is the approach to handling instructions that are not classically virtualizable and cannot be supported via trap and emulate. Full virtualization approaches must detect when the guest OS is attempting to use them and rewrite them to safe instruction sequences at runtime. In contrast, paravirtualization allows the guest OS to be rewritten to not use such instructions, which simplifies the runtime process of providing the virtual machine abstraction.

Another example is how the guest OS interacts with the MMU. In full virtualization, attempts to manipulate the MMU must trap through the host OS (since the guest OS is running without kernel privilege) and be handed to the virtual machine monitor which checks the operation for safety and updates its internal state if appropriate. In paravirtualization, the guest OS is rewritten to use an interface provided by the hypervisor to update the MMU, which can reduce the overhead of the process.

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

Solution:

We discussed two different formulations of Amdahl's Law:

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

—or—

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

We also had our corollary to Amdahl's Law:

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

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

(Note that the same question appeared on the 2013 final exam.)

6. (5 points) Recall that in RAID level 1 (RAID 1) array, both drives store identical contents. (Assume the drives are spinning disks.) First, explain why you would expect to see a significant performance difference between reads and writes to and from a RAID 1 array (3 points). Second, describe how to coordinate RAID 1 reads to further improve performance (2 points).

Solution:

Because both drives store identical contents, RAID 1 reads can come from *either* mirror, allowing both disks to be processing separate reads simultaneously. In contrast, writes *must* complete on *both* disks before they can be considered completed. This accounts for the performance difference.

A simple coordination strategy monitors both disks and uses some metric to determine which disk to send each read to. One way to do this is to monitor the head position of both disks and send the read to the disk where the head has the shortest distance to travel to perform the read. However, this may not work given that if operations are queued at the disk the heads may be repositioned several time before the read under consideration is performed. So it may be more appropriate to look at the location of the *last* read in the disk queue, assuming it processes them in FIFO order. A simpler approach just monitors the read queue length on both disks and sends the read to the disk with the shorter queue—producing a simple form of load balancing. We'll accept anything here that uses information from both disks and seems sane.

VPN	SLT
0	
1	450
2	
3	
4	
5	120
6	
7	
8	
9	320

(a) **TLT**

VPN	PPN
0	31
1	
2	
3	72
4	62
5	
6	40
7	43
8	
9	

(b) **SLT at 320**

VPN	PPN
0	94
1	
2	
3	10
4	13
5	
6	8
7	
8	
9	

(c) **SLT at 120**

VPN	PPN
0	9
1	84
2	
3	39
4	
5	
6	14
7	10
8	
9	

(d) **SLT at 450**

Table 1: **Process Page Tables**

7. (5 points) The `HappierStudent` architecture has byte-addressable memory with 10-byte virtual pages. The `HappierOS` operating system uses two-level pages tables to support 1000-byte virtual address spaces by dividing the virtual page number into a 10-byte second-level index and using the rest as the top-level index.

   First, given the virtual address `465`, identify the virtual page number, the offset, and the first and second level page table indices (1 point).

   Second, given the top-level page table (TLT) and second-level page tables (SLT) above for the currently-running process provided above, indicate the result of the following four virtual to physical page translations (1 point each).

Solution:

For `465`, the virtual page number is `46`, the offset is `5`, and the first and second level page indices are `4` and `6`.

Translations follow:

1. `182` $\rightarrow$ Invalid (no entry in second-level table)
2. `543` $\rightarrow$ `133`
3. `901` $\rightarrow$ `311`
4. `802` $\rightarrow$ Invalid (no entry in top-level table)

(Note that a version of this question with identical page tables but different virtual addresses appeared on the 2015 midterm. However, because few students answered it despite it being quite straightforward, it reappeared on the final exam.)

Medium Answer

8. (20 points) The Exobrowser v. The Exokernel

James Mickens has described his Atlantis browser as an "exokernel browser"—which we will (without permission) shorten to the exobrowser. This terminology reflects that Atlantis shares design principles from the exokernel OS design proposed and implemented by the Dawson Engler, Frans Kaashoek, and other members of the MIT Parallel and Distributed Operating Systems (PDOS) group, in another research paper we read this semester.

First, describe how the exokernel interface differs from the traditional monolithic OS interface (5 points). Second, describe how the exobrowser interface differs from the traditional monolithic browser interface (5 points). Finally, pick one of the design goals that inspired the original exokernel OS design. Describe it (5 points), and discuss how it translates to the exobrowser (5 points).

Solution:

The differences between the exokernel and exobrowser interfaces and the monolithic kernel and monolithic browser interfaces are similar. In both cases, the monolithic version mixes *providing abstractions* with *multiplexing resources*. In the case of the kernel, the abstractions are things like threads, address spaces, and files. In the case of the browser, its interface forces it to provide abstractions the document-object model (DOM) tree, markup (HTML) and layout (CSS) languages, the complete Turing-complete Javascript programming language, and things like the HTML `video` tag which embed rich content into web pages.

For browsers, the idea of abstractions is a bit less clear, but you would have received credit if you pointed out that this change also reduces the kernel or browser interface significantly, pushing more functionality (and complexity) into libraries that now run on top. But of course this also enables more flexibility. Exokernels require that libraries OSes implement things like address spaces, but this allows those libraries to implement them however they want. Similarly, the exobrowser requires that pages specify their own HTML (or whatever markup) parser, but allows the page to know *exactly* how the parsing will take place, or implement its own version of the DOM tree if appropriate.

Here's a specific example of the difference in each case:

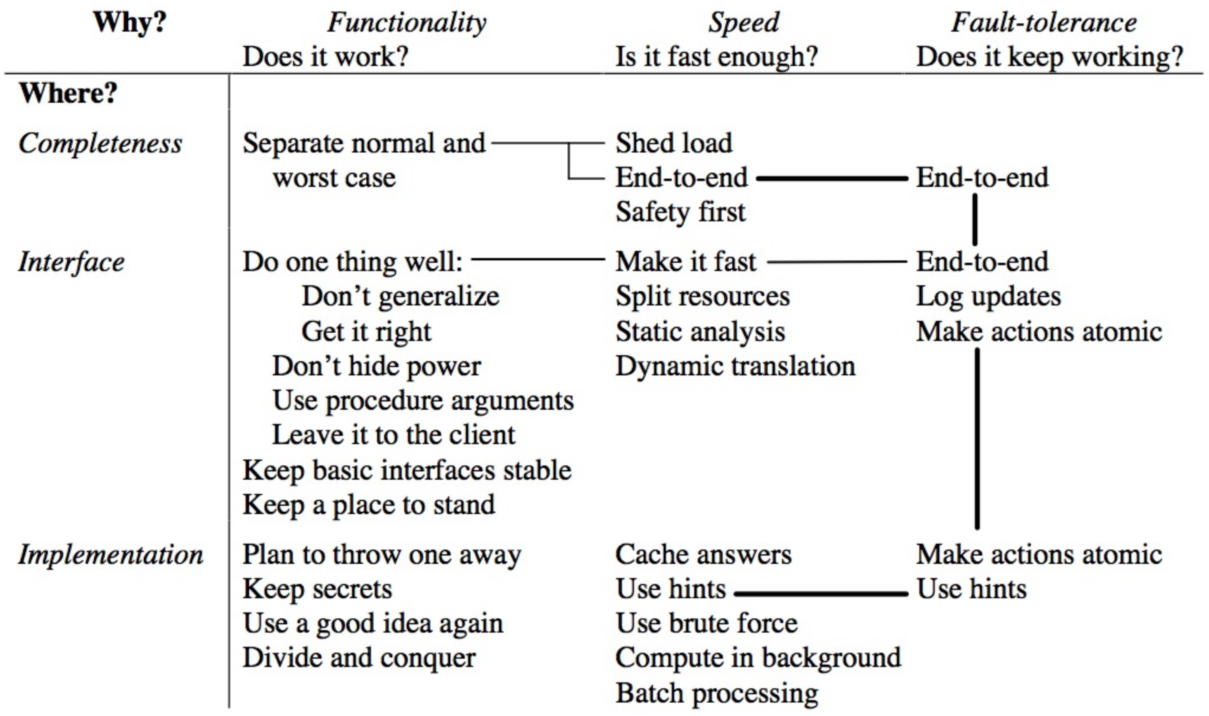
- Monolithic kernels both protect process pages against use by other processes (multiplexing) but also provide a standardized address space abstraction (abstraction). In contrast, the exokernel only ensures that library OSes cannot access each others pages. All other details of virtual memory management, including maintaining page tables and determining address space layout, are moved from the kernel into the library OS.

- Monolithic browsers handle both markup parsing (the process of determining how to convert HTML into on-screen pixels) and rendering (drawing those pixels). In contrast, the exobrowser only provides an interface for library layout engines to draw pixels to the part of the browser display that they control—which also prevents multiple pages from invading each others' visual space. All the details of HTML parsing are left to the page's own parsing engine.

Addressing the second part of the problem, here are a few examples of solutions we would accept based on the design goals presented in class. Recall that the exokernel claimed that fixed abstractions:

- **Limit the functionality of applications.** This probably has the most natural mapping to the exobrowser. One of the central exokernel arguments is that there are opportunities for novel and app-specific features that are difficult or impossible to provide layered on top of traditional OS abstractions. For the exobrowser, this could mean the inability to implement something like a replacement for HTML, a better way of structuring page content to provide interactivity, or a new dynamic page programming language other than Javascript. Or, as the Atlantis paper points out, just the ability to safely shim the `innerHTML` DOM object function to prevent Javascript injection attacks.
- **Hurt application performance.** The exokernel makes the argument that there is no way to implement general OS abstractions in a way that provides good performance for every application—the same page replacement algorithm that works well for one process may work extremely poorly for another. The translation here to the browser is to things like the DOM, a general-purpose abstraction that the browser essentially forces on all pages. Maybe I don't want to handle certain DOM events to avoid the overhead of them firing and hitting empty handlers. Maybe I want to distribute a "compiled" versions of my webpage that avoids all general-purpose HTML and CSS processing. Maybe the page knows exactly which parts its content should be rendered first, and can use this information to reduce the latency to the point where the page is useful. None of this is possible using today's monolithic browsers.
- **Hide useful information from applications.** Finally, the exokernel design makes the claim that applications can benefit from low-level hardware details that are normally hidden from them. For example, knowing (and controlling) exactly where my pages are put on disk might improve the performance of an I/O-heavy database process. Here the parallel with the exobrowser is a bit harder to make, but you could argue that by using Atlantis's low-level networking support it might be possible to expose helpful information about network performance.

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



Why?	*Functionality* Does it work?	*Speed* Is it fast enough?	*Fault-tolerance* Does it keep working?
Where?			
Completeness	Separate normal and worst case	Shed load End-to-end Safety first	End-to-end
Interface	Do one thing well: Don't generalize Get it right Don't hide power Use procedure arguments Leave it to the client Keep basic interfaces stable Keep a place to stand	Make it fast Split resources Static analysis Dynamic translation	End-to-end Log updates Make actions atomic
Implementation	Plan to throw one away Keep secrets Use a good idea again Divide and conquer	Cache answers Use hints Use brute force Compute in background Batch processing	Make actions atomic Use hints

Figure 1: **Summary of the Hints**

Long Answer

9. (25 points) Hints for Computer System Design

Choose five of Butler Lampson's hints for computer system design shown in Figure 1. For each, describe the hint (2 points) and give an example of how to use it—possibly drawn from OS/161 (3 points).

Solution:

For a complete solution, see the paper. But for fun, here are some examples (descriptions omitted) drawn from OS/161:

- **Separate normal and worst case.** The MIPS R3000 provides separate TLB (normal case) and general (worst case) exception handlers.
- **Plan to throw one away.** And you might as well make it simple. Before implementing two-level page tables, try a linked list, or a list of segments. Don't overdesign the first draft.
- **Use brute force.** Particularly for the first implementation. Don't make things fast until they need to be. Apply Amdahl's Law. Unsorted linked list page tables are easier to maintain and not a performance problem until you can demonstrate that the overhead of page translation is dominating.

- **Do one thing well.** Write an interface just for manipulating your page tables. Test it. Then use it as part of a complete ASST3 solution.
- **Shed load.** When you run out of swap, start failing calls to `fork` or `exec`.
- **Use hints.** A page being mapped in the TLB is a hint that it isn't safe to evict. Without holding a lock, a page looking availabtle in the coremap is a hint that it may be after the coremap lock is acquired.
- **Compute in the background.** Writing pages to disk when the system is idle makes swap out much faster.
- **Cache answers.** Use a cache. Cache's are everywhere. The TLB is a cache. The buffer cache is a cache. Saving process TLB entries across context switches is an example of using a cache.

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

10. (25 points) Cross-Device I/O Sharing

Today, a growing number of users interact with multiple personal computing devices: smartphones, tablets, laptops, and desktops. In this multi-device world, the single-device silos that traditional operating system designs were intended to operate seem increasingly obsolete and can prevent seemingly natural cross-device interactions. To illustrate this, consider the following two examples:

- Alice has music playing on her smartphone. When she sits down at her laptop, she wants the music to seamlessly begin playing from her laptop speakers.
- Bob is playing a racing game on his tablet. But instead of controlling it by tilting the tablet, which disrupts his view of the display, he wants to control the tablet game by tilting his smartphone.

To support such scenarios, let's consider how the OS may be able to provide a common platform for cross-device I/O sharing.

To begin, observe that operating systems such as Linux typically reuse the file abstraction to expose many different types of I/O devices to processes. For example, to play music through the sound card the music player would write to the pseudo-file `/dev/audio` and to read gyroscope (tilt) readings the game would read from the pseudo-file `/dev/gyroscope`. Like other places where Linux reuses the file abstraction, there are no contents stored on disk for these pseudo-files. Instead, file operations are converted into the necessary communication with the appropriate device.

Given this information, first describe a design for an OS cross-device I/O sharing system that would allow applications to make use of I/O devices on other nearby devices (10 points). Second, explain why it might be preferable in this case to modify the OS, rather than implement per-application solutions (5 points).

Third, imagine that you were presenting your design to a skeptical audience[1]. Discuss two aspects of your approach that you would need to evaluate to demonstrate that this was a workable solution (5 point each).

Solution:

This examine question was inspired by the Rio system built by Lin Zhong's systems group at Rice University, so for a complete description of the solution please refer to this paper. this paper. There are many interesting details that our simplified problem and solution gloss over.

The key here was to use the hint provided, namely that "operating systems such as Linux typically reuse the file abstraction to expose many different types of I/O devices to processes." This kind of thin interface is sometimes called a "narrow waist", and it gives us a way to support a bunch of different types of devices using a few common mechanisms.

[1]Let's say, a future boss. Or Linus Torvalds.

The design goal is for *unmodified* applications running on Machine A to be able to use a variety of different devices on Machine B *as if* they were local devices. (We'll use Machine to refer to the two computers to avoid colliding with the term "device", which we are using to refer to things like audio cards and gyroscopes.) The way to do this is as follows:

1. **Machine B's local devices need to be exposed to apps running on Machine A.** This may require some local configuration. For example, the user may have to configure both devices so that, when they are nearby, Machine A can view Machine B's devices as if they were local files. As an example, Machine A might see Machine B's audio card `/dev/audio` as `/dev/machineb/audio`. And when configured to use that device, voila, music played by the player on Machine A emerges from the speakers on Machine B.
2. **Trap reads and writes to and from device "files" at the system call interface on Machine A.** For example, when the music player writes to what looks like a local device (`/dev/machineb/audio`), the write is trapped and the write operation, its arguments and its contents are saved in preparation for transmission to Machine B. In the case of the game, the read operation and its arguments are saved. (Note that when reading and writing to devices, some parts of the file abstraction break down. For example, for the microphone and gyroscope the idea of positioning a read or write doesn't make sense, and so neither does `lseek`.)
3. **Transfer the operation and content (in case of writes) from Machine A to Machine B.** This is fairly straightforward, but the latency of this step of the process does depend on the network quality between the two devices.
4. **"Replay" the file operations transferred from Machine A on Machine B on (now) local devices.** This is the dual of the first step. In the case of the microphone, Machine B performs the write request from Machine A on its local device. In the case of the gyroscope, Machine B performs a read on its local device and prepares the data for transmission in the next step.
5. **Return the results and content (in the case of reads) from Machine B to Machine A, again over the network.** Note that our solution has added *two* network transfers to what used to be a local file operation.

The reason why it is preferable to modify the OS is that it allows us to access by unmodified applications to a variety of different types of devices between any two devices running our modified OS. Option A is to wait for both music player and game vendors to provide this capability, and then have to install software updates for both apps on both devices. Option B is to install a single OS upgrade and then have access to this feature for a variety of apps even without developer support. The way that Linux-like operating systems expose devices as files is what makes this possible, but it's the ability to easily support unmodified applications that makes it attractive.

For this to work, there are several requirements that would need to be evaluated. Here's what we came up with, but you may have other ideas:

- **Development burden.** Ideally this should make it easy to support a bunch of devices in a single way, and produce minimal changes to the stock Linux operating system. This is particularly important given that it may be possible or necessary to also support other types of Linux-like operating systems. If major changes to Linux are required, or if the changes scale with the number of devices we want to support, then this could be a nonstarter.
- **Performance.** You don't need to understanding any of the networking details to assume that introducing the network is going to add latency and overhead to what used to be local device operations. For the music player this may be tolerable, as a bit of lag for commands is acceptable as long as the music continues to play smoothly. For the game, however, too much delay could render a fast-pace game unplayable. And a very high bandwidth sensor like a camera could just generate too much data for the network to keep up.
- **App compatibility.** Changes to the behavior of what look like local devices could also cause apps to break or misbehave—for example, if they rely on timing guarantees that the network makes it difficult to provide.