CSE 421/521 Final Exam Solutions

—SOLUTION SET—

09 May 2016

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) One day in class GWA ate

   √ **something orange.** √ **oranges.** √ **something that Ali videotaped him eating.** √ **a fruit of the citrus variety.**

   (b) Significant differences between file systems include everything *except*

   ◯ on-disk layout. √ **reliably storing Jinghao's data.** ◯ data structures. ◯ crash recovery mechanisms.

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

   ◯ Running `gmake` √ **Reducing false sharing in the cache by rearranging in-memory data structures** ◯ Rewriting applications ◯ Asking Carl

   (d) Flash drives present none of the layout difficulties typical to spinning disks.

   ◯ True √ **False**

   (e) On `ext4` inodes are *not* stored

   ◯ in groups. √ **right next to the file contents.** ◯ in fixed locations.

   (f) 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 ◯ Gela doesn't like it ◯ It's shared by multiple processes

   (g) When your performance data has outliers, you should *not*

   √ **hide them by exclusively using summary statistics.** ◯ inspect them carefully with Xu. ◯ rexamine your mental model of the system. ◯ ensure that they are not due to bugs in your simulator.

   (h) Log-structured file systems provide better performance.

   ◯ True √ **For some workloads** ◯ False

   (i) Which of the following is *not* a useful approach to improving performance?

   ◯ Talking to Yousuf about choosing an appropriate benchmark. √ **Improving the parts of your code that you just know are slow.** ◯ Analyzing data from experiments to identify bottlenecks. ◯ Developing a new simulator to improve reproducibility.

   (j) What is a company known for container virtualization?

   ◯ Virtualbox ◯ Vagrant √ **Docker** ◯ Haseley Inc.

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) 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.

(This question appeared previously on the 2015 final exam.)

3. (5 points) List and explain three of Butler Lampson's hints for improving computer system performance (1 point per hint, 1 point per explanation up to 5 points total).

Solution: This was a watered-down version of a long-answer question that appeared on the 2015 final exam. As such, briefer explanations were accepted.

For a complete solution, see the paper. Here are just a few examples:

- **Separate normal and worst case.** Don't make everything go down a slow path if there is a frequent fast solution that applies in many cases.
- **Plan to throw one away.** Do a simple implementation first and plan to rewrite it once you understand the problem better.
- **Use brute force.** Don't make things fast until they need to be.
- **Do one thing well.** Refactor code into simpler functions that you can more easily test and understand. Then build them into a larger more complex solution.
- **Compute in the background.** Try to move slow operations into the background where users won't notice them.
- **Cache answers.** Use a cache! Sometimes this is now known as memoization or dynamic programming.

4. (5 points) 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.**

   However, Amdahl's Law also has an important corollary. First, state it (3 points), and then explain how it also guides the process of performance improvement (2 points).

Solution: The important corollary to Amdahl's Law was that the longer you optimize one part of the system the less likely it is that you are still working on the right problem. As one part gets faster, other parts that were not the bottleneck when you started end up becoming the bottleneck and further improvements to the part that you are optimizing have diminished returns. As a result, it is important to periodically step back and restart the benchmarking and performance improvement process to ensure that you are always working on the most important part of the code.

5. (5 points) Explain the difference between placing the buffer cache above the file system interface or below it. What interface must the cache support at each level (2 points)? What is cached (and not cached) in each case (2 points)? Describe one operation with a significant performance difference in each case and explain why this occurs (1 point).

Solution: The buffer cache must support the interface of whatever is directly beneath it. Above the file system it must support the file system interface—`open`, `close`, `read`, `write`, etc. Below it it must support the disk block interface—just reading and writing blocks.

What the buffer cache can cache is what is returned from the interface below it. Above the file system it can cache contents passed to (or returned from) `read` and `write`. Below the file system it can cache the disk blocks passed to (or returned from) the disk block interface.

Because the buffer cache above the file system cannot cache metadata— including inodes, directory contents, and other file system on-disk data structures—operations such as `open` that mainly utilize metadata will miss the cache and perform more poorly on a cache located above the file system. (Note that this question was released prior to the final exam.)

6. (5 points) Describe how file system journaling works:

   - What is written to the journal (2 points)?
   - How is the journal used after a crash to quickly return the file system to a consistent state (3 points)?

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

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

Checkpoints are written to the journal periodically and indicate that all changes recorded previously have been written to disk. When a crash occurs, the file system uses the journal to recover a consistent state. It does this by scanning the journal starting at the latest checkpoint and looking for uncommitted operations: those that are written to the journal but not to disk. Once all uncommitted operations have been flushed to the disk, the file system should be in a consistent state and can resume operation. (This question first appeared on the 2012 final exam.)

7. (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.

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.)

Long Answer

9. (25 points) Supporting Heterogeneous Cores

An increasing number of computer systems—from smartphones to servers—now feature multiple *heterogeneous cores* with different power-performance tradeoffs. For example, ARM now has many systems that use their `big.LITTLE` architecture which combines one or more fast and high-power cores with one or more slower but lower-power cores. (While this is not important to answering the question, the reason for this approach is that how the core is made has an impact on how much power it consumes—even in cases where the energy-performance tradeoff can be adjusted at runtime by changing the operating frequency.)

Operating systems with heterogeneous cores creates new challenges and opportunities for OS design. In this question we ask you to consider a two-core heterogeneous system with the following properties:

1. **Performance and energy consumption.** Both cores can scale their speed and power consumption up and down at runtime, but the big core is always faster and always consumes more energy than the small core. Put another way, even when the big core is running at its *slowest* frequency it is almost equivalent to the small core running at its *fastest* frequency.
2. **Instruction Set Compatibility.** All of the user-mode instructions that can be run on the small core can be run on the big core (more quickly), but all of the user-mode instructions that can be run on the big core *cannot* be run on the small core. However, those instructions will generate an exception if they are run on the small core. (You can assume all kernel-mode instructions work identically on both cores.)
3. **Memory Access.** The big core has access to all of the hardware memory, but the small core can only see the first 512 MB of available hardware memory. Unlike the instruction set case above, attempts to access invalid hardware memory address on the small core will result in fatal exceptions that cause both processors to reset, so they must be avoided by the OS.

Describe how to design an OS that allows multithreaded application processes to effectively and efficiently use both big and small cores. First, explain two of the OS design challenges that this architecture creates compared with a typical homogeneous multicore share memory system (5 points each). Second, describe how to address each challenge through changes to the OS but *without* changing compiled applications (5 points each). Finally, describe changes to applications themselves or the OS API that might help improve performance or energy efficiency on this heterogeneous system (5 points).

Rubric:

As embedded in the question.

- **+5 points** per well-explained design challenge.
- **+5 points** per well-addressed challenge through OS changes.
- **+5 points** for a reasonable change to the applications or OS API to better support this system.

Solution:

This question was inspired by Lin Zhong's work on a system called K2, which provides a mobile operating system for heterogeneous coherence domains. Take a look at that paper for a much more complete solution to a related problem.

There are several challenges and workarounds that you could have proposed, related to both the inconsistent instruction set and view of memory between the little and big cores.

1. **Scheduling to save energy but avoid expensive cross-core switching.** There are cases (such as background operation) when you want to be able to run threads on the little core. This is always safe to do, but it can trigger transitions to the big core when the thread tries to run instructions that require the big core. These transitions are really no different that a context switch, but in this case one created by an exception rather than a timer interrupt or blocking system call. You could assume that the register architecture of the two cores was the same, meaning that whenever a thread running on the little core hits an "needs the big core" exception, it is stopped, the big core is signaled (and awakened if necessary), and the thread is restarted on the big core where that instruction can complete correctly. The tradeoff here is that to save energy (again assuming background operation) you want to run things on the little core as much as possible, but premature transitions back to the little core can cause a lot of switching back and forth, while continuing to run on the big core wastes energy. (There also may be some startup cost to waking up the big core, wasting both time and energy.) It may be possible for the OS to do some profiling to determine the likelihood that a given code path will lead to an instruction requiring the big core and avoid migrating the thread back to the little core when executing in these code regions. But this is certainly something that the scheduler needs to consider, and you would get credit for pointing out this challenge and that the scheduler needs to somehow make this tradeoff intelligently.
2. **Arranging memory mappings to avoid having to active the big core.** Note that the little core cannot access all of memory. As a result, when accessing any parts of its address space that are mapped outside of the 512 MB visible to the little core the thread must be moved to the big core, even if it is executing instructions that it could continue to execute on the small core. Ideally the OS would avoid these

unnecessary transitions when trying to save energy by setting up virtual memory mappings for each process that tried to put things in the memory available to the small core, but then what do you do with the rest of memory? One approach is to use that area for application code that uses big-core-only instructions—for example, code pages that have a large number of big-core-only instructions can be safely put there, since they are likely to trigger the activation of the big core anyway. But this is another new design challenge for the OS to consider when establishing virtual memory mappings. You would receive credit for noting that the virtual memory mappings are now linked to the core that is running and trying to find an intelligent way to limit the number of unnecessary activations of the big core that are driven entirely by needing access to memory that the little core cannot see—rather than for other reasons.

3. **Making smooth performance and energy consumption transitions between cores.** The goal of having heterogeneous cores in the first place is to create the illusion of a single core—but one with a energy-performance range that cannot be achieved using a single architectural design process. Unfortunately, given the incompatibilities described above ensuring smooth transitions between the two cores is non-trivial. In particular, the OS may want to consider ensuring that tasks that operate around the transition point between the two cores don't bounce frequently back and forth between them, since this creates a lot of extra overhead. A better solution is to run a task at the lowest frequency level (on the big core) or highest (on the little core) for a while before making the decision to transition, to ensure that the task really needs the faster (and less energy efficiency) or more energy efficient (and slower) settings provided by the other core. You would have received credit for pointing out the design goal of uniting the two cores but the problems created by the "transition zone" between them.
4. **Dividing OS functionality between cores.** Operating systems for this kind of hardware also need to think carefully about what OS services and functionality can be run on each core. For example, if OS code or data structures are stored outside of the 512 MB of RAM visible to the little core, those tasks cannot be run on the little core. 512 MB isn't a huge amount of memory, and packing all of the kernel data structures down in that region limits the amount of memory available to user tasks running on the small core, so this is not a hypothetical concern. If, for example, process page tables are stored out of the 512 MB little core region, then some page faults may need to be actually handled by the big core even if the resulting pages are loaded into the 512 MB little core region. At this point it might be simpler just to move the faulting thread to the big core to execute, although there are energy considerations to this approach. You would have received credit for noting that OS functionality must also be carefully split and having some idea when OS tasks would need to be migrated between cores.

As far as changes to the OS API or applications, there are a lot of different things to

try. Programmer hints or annotations may indicate what threads should be pinned to the small core and compiled without use of big core instructions, or this may be something that can be determined at compile time or with some amount of pre-deployment profiling. Alternatively, changes to the OS API for creating threads could allow processes to again designate that a particular thread should only run on the little (or big) core. If a thread is pinned to the little core, any big-core instructions would generate not a transition to the big core but a fatal exception, allowing the system to enforce the programming model. Essentially, any changes to the programming language, OS API, or compilation process to given the OS more information about what core to run various tasks and threads on would be useful. But you may have come up with other good ideas here as well.

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

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. . .[1]

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)[2].

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!

[1] Thanks to Katelin Bailey, Luis Ceze, Steven D. Gribble and Henry M. Levy for inspiring this question.

[2] You can assume that we continue to use processor caches.

Rubric:

As embedded in the question:

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

Solution:

(This question first appeared on the 2014 final exam.)

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 file system 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, file systems 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 file systems. Part of this richness, obviously, is because file system 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 file system naming.** Another fundamental difference between memory and file systems 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 name space*—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, file systems provide a *global name space*: `/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 file systems and files are dead and over anyway, and all data transfer in the future will take place through the cloud. Who knows.