A

A Dialogue on Virtual Machine Monitors

Student: *So now we're stuck in the Appendix, huh?*

Professor: *Yes, just when you thought things couldn't get any worse.*

Student: *Well, what are we going to talk about?*

Professor: *An old topic that has been reborn:* ***virtual machine monitors****, also known as* ***hypervisors****.*

Student: *Oh, like VMware? That's cool; I've used that kind of software before.*

Professor: *Cool indeed. We'll learn how VMMs add yet another layer of virtualization into systems, this one beneath the OS itself! Crazy and amazing stuff, really.*

Student: *Sounds neat. Why not include this in the earlier part of the book, then, on virtualization? Shouldn't it really go there?*

Professor: *That's above our pay grade, I'm afraid. But my guess is this: there is already a lot of material there. By moving this small aside on VMMs into the appendix, a particular instructor can choose whether to include it or skip it. But I do think it should be included, because if you can understand how VMMs work, then you really understand virtualization quite well.*

Student: *Alright then, let's get to work!*

B

Virtual Machine Monitors

B.1 Introduction

Years ago, IBM sold expensive mainframes to large organizations, and a problem arose: what if the organization wanted to run different operating systems on the machine at the same time? Some applications had been developed on one OS, and some on others, and thus the problem. As a solution, IBM introduced yet another level of indirection in the form of a **virtual machine monitor** (**VMM**) (also called a **hypervisor**) [G74].

Specifically, the monitor sits between one or more operating systems and the hardware and gives the illusion to each running OS that it controls the machine. Behind the scenes, however, the monitor actually is in control of the hardware, and must multiplex running OSes across the physical resources of the machine. Indeed, the VMM serves as an operating system for operating systems, but at a much lower level; the OS must still think it is interacting with the physical hardware. Thus, **transparency** is a major goal of VMMs.

Thus, we find ourselves in a funny position: the OS has thus far served as the master illusionist, tricking unsuspecting applications into thinking they have their own private CPU and a large virtual memory, while secretly switching between applications and sharing memory as well. Now, we have to do it again, but this time underneath the OS, who is used to being in charge. How can the VMM create this illusion for each OS running on top of it?

THE CRUX:
HOW TO VIRTUALIZE THE MACHINE UNDERNEATH THE OS
The virtual machine monitor must transparently virtualize the machine underneath the OS; what are the techniques required to do so?

B.2 Motivation: Why VMMs?

Today, VMMs have become popular again for a multitude of reasons. Server consolidation is one such reason. In many settings, people run services on different machines which run different operating systems (or even OS versions), and yet each machine is lightly utilized. In this case, virtualization enables an administrator to **consolidate** multiple OSes onto fewer hardware platforms, and thus lower costs and ease administration.

Virtualization has also become popular on desktops, as many users wish to run one operating system (say Linux or Mac OS X) but still have access to native applications on a different platform (say Windows). This type of improvement in **functionality** is also a good reason.

Another reason is testing and debugging. While developers write code on one main platform, they often want to debug and test it on the many different platforms that they deploy the software to in the field. Thus, virtualization makes it easy to do so, by enabling a developer to run many operating system types and versions on just one machine.

This resurgence in virtualization began in earnest the mid-to-late 1990's, and was led by a group of researchers at Stanford headed by Professor Mendel Rosenblum. His group's work on Disco [B+97], a virtual machine monitor for the MIPS processor, was an early effort that revived VMMs and eventually led that group to the founding of VMware [V98], now a market leader in virtualization technology. In this chapter, we will discuss the primary technology underlying Disco and through that window try to understand how virtualization works.

B.3 Virtualizing the CPU

To run a **virtual machine** (e.g., an OS and its applications) on top of a virtual machine monitor, the basic technique that is used is **limited direct execution**, a technique we saw before when discussing how the OS virtualizes the CPU. Thus, when we wish to "boot" a new OS on top of the VMM, we simply jump to the address of the first instruction and let the OS begin running. It is as simple as that (well, almost).

Assume we are running on a single processor, and that we wish to multiplex between two virtual machines, that is, between two OSes and their respective applications. In a manner quite similar to an operating system switching between running processes (a **context switch**), a virtual machine monitor must perform a **machine switch** between running virtual machines. Thus, when performing such a switch, the VMM must save the entire machine state of one OS (including registers, PC, and unlike in a context switch, any privileged hardware state), restore the machine state of the to-be-run VM, and then jump to the PC of the to-be-run VM and thus complete the switch. Note that the to-be-run VM's PC may be within the OS itself (i.e., the system was executing a system call) or it may simply be within a process that is running on that OS (i.e., a user-mode application).

We get into some slightly trickier issues when a running application or OS tries to perform some kind of **privileged operation**. For example, on a system with a software-managed TLB, the OS will use special privileged instructions to update the TLB with a translation before restarting an instruction that suffered a TLB miss. In a virtualized environment, the OS cannot be allowed to perform privileged instructions, because then it controls the machine rather than the VMM beneath it. Thus, the VMM must somehow intercept attempts to perform privileged operations and thus retain control of the machine.

A simple example of how a VMM must interpose on certain operations arises when a running process on a given OS tries to make a system call. For example, the process may be trying to call `open()` on a file, or may be calling `read()` to get data from it, or may be calling `fork()` to create a new process. In a system without virtualization, a system call is achieved with a special instruction; on MIPS, it is a **trap** instruction, and on x86, it is the `int` (an interrupt) instruction with the argument `0x80`. Here is the open library call on FreeBSD [B00] (recall that your C code first makes a library call into the C library, which then executes the proper assembly sequence to actually issue the trap instruction and make a system call):

```
open:
    push    dword mode
    push    dword flags
    push    dword path
    mov     eax, 5
    push    eax
    int     80h
```

On UNIX-based systems, `open()` takes just three arguments: `int open(char *path, int flags, mode_t mode)`. You can see in the code above how the `open()` library call is implemented: first, the arguments get pushed onto the stack (`mode, flags, path`), then a 5 gets pushed onto the stack, and then `int 80h` is called, which transfers control to the kernel. The 5, if you were wondering, is the pre-agreed upon convention between user-mode applications and the kernel for the `open()` system call in FreeBSD; different system calls would place different numbers onto the stack (in the same position) before calling the trap instruction `int` and thus making the system call[1].

When a trap instruction is executed, as we've discussed before, it usually does a number of interesting things. Most important in our example here is that it first transfers control (i.e., changes the PC) to a well-defined **trap handler** within the operating system. The OS, when it is first starting up, establishes the address of such a routine with the hardware (also a privileged operation) and thus upon subsequent traps, the hardware

[1]Just to make things confusing, the Intel folks use the term "interrupt" for what almost any sane person would call a trap instruction. As Patterson said about the Intel instruction set: "It's an ISA only a mother could love." But actually, we kind of like it, and we're not its mother.

Process	Hardware	Operating System
1. Execute instructions (add, load, etc.)		
2. System call: Trap to OS		
	3. Switch to kernel mode; Jump to trap handler	
		4. In kernel mode; Handle system call; Return from trap
	5. Switch to user mode; Return to user code	
6. Resume execution (@PC after trap)		

Table B.1: **Executing a System Call**

knows where to start running code to handle the trap. At the same time of the trap, the hardware also does one other crucial thing: it changes the mode of the processor from **user mode** to **kernel mode**. In user mode, operations are restricted, and attempts to perform privileged operations will lead to a trap and likely the termination of the offending process; in kernel mode, on the other hand, the full power of the machine is available, and thus all privileged operations can be executed. Thus, in a traditional setting (again, without virtualization), the flow of control would be like what you see in Table B.1.

On a virtualized platform, things are a little more interesting. When an application running on an OS wishes to perform a system call, it does the exact same thing: executes a trap instruction with the arguments carefully placed on the stack (or in registers). However, it is the VMM that controls the machine, and thus the VMM who has installed a trap handler that will first get executed in kernel mode.

So what should the VMM do to handle this system call? The VMM doesn't really know **how** to handle the call; after all, it does not know the details of each OS that is running and therefore does not know what each call should do. What the VMM does know, however, is **where** the OS's trap handler is. It knows this because when the OS booted up, it tried to install its own trap handlers; when the OS did so, it was trying to do something privileged, and therefore trapped into the VMM; at that time, the VMM recorded the necessary information (i.e., where this OS's trap handlers are in memory). Now, when the VMM receives a trap from a user process running on the given OS, it knows exactly what to do: it jumps to the OS's trap handler and lets the OS handle the system call as it should. When the OS is finished, it executes some kind of privileged instruction to return from the trap (**rett** on MIPS, `iret` on x86), which again bounces into the VMM, which then realizes that the OS is trying to return from the trap and thus performs a real return-from-trap and thus returns control to the user and puts the machine back in user mode. The entire process is depicted in Tables B.2 and B.3, both for the normal case without virtualization and the case with virtualization (we leave out the exact hardware operations from above to save space).

Process	Operating System
1. System call: Trap to OS	
	2. OS trap handler: Decode trap and execute appropriate syscall routine; When done: return from trap
3. Resume execution (@PC after trap)	

Table B.2: **System Call Flow Without Virtualization**

Process	Operating System	VMM
1. System call: Trap to OS		
		2. Process trapped: Call OS trap handler (at reduced privilege)
	3. OS trap handler: Decode trap and execute syscall; When done: issue return-from-trap	
		4. OS tried return from trap: Do real return from trap
5. Resume execution (@PC after trap)		

Table B.3: **System Call Flow with Virtualization**

As you can see from the figures, a lot more has to take place when virtualization is going on. Certainly, because of the extra jumping around, virtualization might indeed slow down system calls and thus could hurt performance.

You might also notice that we have one remaining question: what mode should the OS run in? It can't run in kernel mode, because then it would have unrestricted access to the hardware. Thus, it must run in some less privileged mode than before, be able to access its own data structures, and simultaneously prevent access to its data structures from user processes.

In the Disco work, Rosenblum and colleagues handled this problem quite neatly by taking advantage of a special mode provided by the MIPS hardware known as supervisor mode. When running in this mode, one still doesn't have access to privileged instructions, but one can access a little more memory than when in user mode; the OS can use this extra memory for its data structures and all is well. On hardware that doesn't have such a mode, one has to run the OS in user mode and use memory protection (page tables and TLBs) to protect OS data structures appropriately. In other words, when switching into the OS, the monitor would have to make the memory of the OS data structures available to the OS via page-table protections; when switching back to the running application, the ability to read and write the kernel would have to be removed.

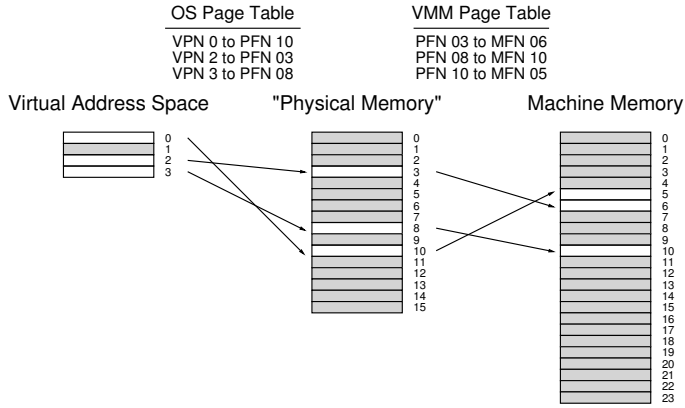


Figure B.1: **VMM Memory Virtualization**

B.4 Virtualizing Memory

You should now have a basic idea of how the processor is virtualized: the VMM acts like an OS and schedules different virtual machines to run, and some interesting interactions occur when privilege levels change. But we have left out a big part of the equation: how does the VMM virtualize memory?

Each OS normally thinks of physical memory as a linear array of pages, and assigns each page to itself or user processes. The OS itself, of course, already virtualizes memory for its running processes, such that each process has the illusion of its own private address space. Now we must add another layer of virtualization, so that multiple OSes can share the actual physical memory of the machine, and we must do so transparently.

This extra layer of virtualization makes "physical" memory a virtualization on top of what the VMM refers to as **machine memory**, which is the real physical memory of the system. Thus, we now have an additional layer of indirection: each OS maps virtual-to-physical addresses via its per-process page tables; the VMM maps the resulting physical mappings to underlying machine addresses via its per-OS page tables. Figure B.1 depicts this extra level of indirection.

In the figure, there is just a single virtual address space with four pages, three of which are valid (0, 2, and 3). The OS uses its page table to map these pages to three underlying physical frames (10, 3, and 8, respectively). Underneath the OS, the VMM performs a further level of indirection, mapping PFNs 3, 8, and 10 to machine frames 6, 10, and 5 respectively. Of course, this picture simplifies things quite a bit; on a real system, there would be V operating systems running (with V likely

Process	Operating System
1. Load from memory: TLB miss: Trap	
	2. OS TLB miss handler: Extract VPN from VA; Do page table lookup; If present and valid: get PFN, update TLB; Return from trap
3. Resume execution (@PC of trapping instruction); Instruction is retried; Results in TLB hit	

Table B.4: **TLB Miss Flow without Virtualization**

greater than one), and thus V VMM page tables; further, on top of each running operating system OS_i, there would be a number of processes P_i running (P_i likely in the tens or hundreds), and hence P_i (per-process) page tables within OS_i.

To understand how this works a little better, let's recall how **address translation** works in a modern paged system. Specifically, let's discuss what happens on a system with a software-managed TLB during address translation. Assume a user process generates an address (for an instruction fetch or an explicit load or store); by definition, the process generates a **virtual address**, as its address space has been virtualized by the OS. As you know by now, it is the role of the OS, with help from the hardware, to turn this into a **physical address** and thus be able to fetch the desired contents from physical memory.

Assume we have a 32-bit virtual address space and a 4-KB page size. Thus, our 32-bit address is chopped into two parts: a 20-bit virtual page number (VPN), and a 12-bit offset. The role of the OS, with help from the hardware TLB, is to translate the VPN into a valid physical page frame number (PFN) and thus produce a fully-formed physical address which can be sent to physical memory to fetch the proper data. In the common case, we expect the TLB to handle the translation in hardware, thus making the translation fast. When a TLB miss occurs (at least, on a system with a software-managed TLB), the OS must get involved to service the miss, as depicted here in Table B.4.

As you can see, a TLB miss causes a trap into the OS, which handles the fault by looking up the VPN in the page table and installing the translation in the TLB.

With a virtual machine monitor underneath the OS, however, things again get a little more interesting. Let's examine the flow of a TLB miss again (see Table B.5 for a summary). When a process makes a virtual memory reference and misses in the TLB, it is not the OS TLB miss handler that runs; rather, it is the VMM TLB miss handler, as the VMM is the true privileged owner of the machine. However, in the normal case, the VMM TLB handler doesn't know how to handle the TLB miss, so it immediately jumps into the OS TLB miss handler; the VMM knows the

Process	Operating System	Virtual Machine Monitor
1. Load from memory TLB miss: Trap		
		2. VMM TLB miss handler: Call into OS TLB handler (reducing privilege)
	3. OS TLB miss handler: Extract VPN from VA; Do page table lookup; If present and valid, get PFN, update TLB	
		4. Trap handler: Unprivileged code trying to update the TLB; OS is trying to install VPN-to-PFN mapping; Update TLB instead with VPN-to-MFN (privileged); Jump back to OS (reducing privilege)
	5. Return from trap	
		6. Trap handler: Unprivileged code trying to return from a trap; Return from trap
7. Resume execution (@PC of instruction); Instruction is retried; Results in TLB hit		

Table B.5: **TLB Miss Flow with Virtualization**

location of this handler because the OS, during "boot", tried to install its own trap handlers. The OS TLB miss handler then runs, does a page table lookup for the VPN in question, and tries to install the VPN-to-PFN mapping in the TLB. However, doing so is a privileged operation, and thus causes another trap into the VMM (the VMM gets notified when any non-privileged code tries to do something that is privileged, of course). At this point, the VMM plays its trick: instead of installing the OS's VPN-to-PFN mapping, the VMM installs its desired VPN-to-MFN mapping. After doing so, the system eventually gets back to the user-level code, which retries the instruction, and results in a TLB hit, fetching the data from the machine frame where the data resides.

This set of actions also hints at how a VMM must manage the virtualization of physical memory for each running OS; just like the OS has a page table for each process, the VMM must track the physical-to-machine mappings for each virtual machine it is running. These per-machine page tables need to be consulted in the VMM TLB miss handler in order to determine which machine page a particular "physical" page maps to, and even, for example, if it is present in machine memory at the current time (i.e., the VMM could have swapped it to disk).

ASIDE: **HYPERVISORS AND HARDWARE-MANAGED TLBS**

Our discussion has centered around software-managed TLBs and the work that needs to be done when a miss occurs. But you might be wondering: how does the virtual machine monitor get involved with a hardware-managed TLB? In those systems, the hardware walks the page table on each TLB miss and updates the TLB as need be, and thus the VMM doesn't have a chance to run on each TLB miss to sneak its translation into the system. Instead, the VMM must closely monitor changes the OS makes to each page table (which, in a hardware-managed system, is pointed to by a page-table base register of some kind), and keep a **shadow page table** that instead maps the virtual addresses of each process to the VMM's desired machine pages [AA06]. The VMM installs a process's shadow page table whenever the OS tries to install the process's OS-level page table, and thus the hardware chugs along, translating virtual addresses to machine addresses using the shadow table, without the OS even noticing.

Finally, as you might notice from this sequence of operations, TLB misses on a virtualized system become quite a bit more expensive than in a non-virtualized system. To reduce this cost, the designers of Disco added a VMM-level "software TLB". The idea behind this data structure is simple. The VMM records every virtual-to-physical mapping that it sees the OS try to install; then, on a TLB miss, the VMM first consults its software TLB to see if it has seen this virtual-to-physical mapping before, and what the VMM's desired virtual-to-machine mapping should be. If the VMM finds the translation in its software TLB, it simply installs the virtual-to-machine mapping directly into the hardware TLB, and thus skips all the back and forth in the control flow above [B+97].

B.5 The Information Gap

Just like the OS doesn't know too much about what application programs really want, and thus must often make general policies that hopefully work for all programs, the VMM often doesn't know too much about what the OS is doing or wanting; this lack of knowledge, sometimes called the **information gap** between the VMM and the OS, can lead to various inefficiencies [B+97]. For example, an OS, when it has nothing else to run, will sometimes go into an **idle loop** just spinning and waiting for the next interrupt to occur:

```
while (1)
  ; // the idle loop
```

It makes sense to spin like this if the OS in charge of the entire machine and thus knows there is nothing else that needs to run. However, when a

ASIDE: **PARA-VIRTUALIZATION**
In many situations, it is good to assume that the OS cannot be modified in order to work better with virtual machine monitors (for example, because you are running your VMM under an unfriendly competitor's operating system). However, this is not always the case, and when the OS can be modified (as we saw in the example with demand-zeroing of pages), it may run more efficiently on top of a VMM. Running a modified OS to run on a VMM is generally called **para-virtualization** [WSG02], as the virtualization provided by the VMM isn't a complete one, but rather a partial one requiring OS changes to operate effectively. Research shows that a properly-designed para-virtualized system, with just the right OS changes, can be made to be nearly as efficient a system without a VMM [BD+03].

VMM is running underneath two different OSes, one in the idle loop and one usefully running user processes, it would be useful for the VMM to know that one OS is idle so it can give more CPU time to the OS doing useful work.

Another example arises with demand zeroing of pages. Most operating systems zero a physical frame before mapping it into a process's address space. The reason for doing so is simple: security. If the OS gave one process a page that another had been using *without* zeroing it, an information leak across processes could occur, thus potentially leaking sensitive information. Unfortunately, the VMM must zero pages that it gives to each OS, for the same reason, and thus many times a page will be zeroed twice, once by the VMM when assigning it to an OS, and once by the OS when assigning it to a process. The authors of Disco had no great solution to this problem: they simply changed the OS (IRIX) to not zero pages that it knew had been zeroed by the underlying VMM [B+97].

There are many other similar problems to these described here. One solution is for the VMM to use inference (a form of **implicit information**) to overcome the problem. For example, a VMM can detect the idle loop by noticing that the OS switched to low-power mode. A different approach, seen in **para-virtualized** systems, requires the OS to be changed. This more explicit approach, while harder to deploy, can be quite effective.

B.6 Summary

Virtualization is in a renaissance. For a multitude of reasons, users and administrators want to run multiple OSes on the same machine at the same time. The key is that VMMs generally provide this service **transparently**; the OS above has little clue that it is not actually controlling the hardware of the machine. The key method that VMMs use to do so is to extend the notion of limited direct execution; by setting up the hard-

TIP: USE IMPLICIT INFORMATION

Implicit information can be a powerful tool in layered systems where it is hard to change the interfaces between systems, but more information about a different layer of the system is needed. For example, a block-based disk device might like to know more about how a file system above it is using it; Similarly, an application might want to know what pages are currently in the file-system page cache, but the OS provides no API to access this information. In both these cases, researchers have developed powerful inferencing techniques to gather the needed information implicitly, *without* requiring an explicit interface between layers [AD+01,S+03]. Such techniques are quite useful in a virtual machine monitor, which would like to learn more about the OSes running above it without requiring an explicit API between the two layers.

ware to enable the VMM to interpose on key events (such as traps), the VMM can completely control how machine resources are allocated while preserving the illusion that the OS requires.

You might have noticed some similarities between what the OS does for processes and what the VMM does for OSes. They both virtualize the hardware after all, and hence do some of the same things. However, there is one key difference: with the OS virtualization, a number of new abstractions and nice interfaces are provided; with VMM-level virtualization, the abstraction is identical to the hardware (and thus not very nice). While both the OS and VMM virtualize hardware, they do so by providing completely different interfaces; VMMs, unlike the OS, are not particularly meant to make the hardware easier to use.

There are many other topics to study if you wish to learn more about virtualization. For example, we didn't even discuss what happens with I/O, a topic that has its own new and interesting issues when it comes to virtualized platforms. We also didn't discuss how virtualization works when running "on the side" with your OS in what is sometimes called a "hosted" configuration. Read more about both of these topics if you're interested [SVL01]. We also didn't discuss what happens when a collection of operating systems running on a VMM uses too much memory.

Finally, hardware support has changed how platforms support virtualization. Companies like Intel and AMD now include direct support for an extra level of virtualization, thus obviating many of the software techniques in this chapter. Perhaps, in a chapter yet-to-be-written, we will discuss these mechanisms in more detail.

References

[AA06] "A Comparison of Software and Hardware Techniques
for x86 Virtualization"
Keith Adams and Ole Agesen
ASPLOS '06, San Jose, California
A terrific paper from two VMware engineers about the surprisingly small benefits of having hardware support for virtualization. Also an excellent general discussion about virtualization in VMware, including the crazy binary-translation tricks they have to play in order to virtualize the difficult-to-virtualize x86 platform.

[AD+01] "Information and Control in Gray-box Systems"
Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau
SOSP '01, Banff, Canada
Our own work on how to infer information and even exert control over the OS from application level, without any change to the OS. The best example therein: determining which file blocks are cached in the OS using a probabilistic probe-based technique; doing so allows applications to better utilize the cache, by first scheduling work that will result in hits.

[B00] "FreeBSD Developers' Handbook:
Chapter 11 x86 Assembly Language Programming"
http://www.freebsd.org/doc/en/books/developers-handbook/
A nice tutorial on system calls and such in the BSD developers handbook.

[BD+03] "Xen and the Art of Virtualization"
Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, Andrew Warfield
SOSP '03, Bolton Landing, New York
The paper that shows that with para-virtualized systems, the overheads of virtualized systems can be made to be incredibly low. So successful was this paper on the Xen virtual machine monitor that it launched a company.

[B+97] "Disco: Running Commodity Operating Systems
on Scalable Multiprocessors"
Edouard Bugnion, Scott Devine, Kinshuk Govil, Mendel Rosenblum
SOSP '97
The paper that reintroduced the systems community to virtual machine research; well, perhaps this is unfair as Bressoud and Schneider [BS95] also did, but here we began to understand why virtualization was going to come back. What made it even clearer, however, is when this group of excellent researchers started VMware and made some billions of dollars.

[BS95] "Hypervisor-based Fault-tolerance"
Thomas C. Bressoud, Fred B. Schneider
SOSP '95
*One the earliest papers to bring back the **hypervisor**, which is just another term for a virtual machine monitor. In this work, however, such hypervisors are used to improve system tolerance of hardware faults, which is perhaps less useful than some of the more practical scenarios discussed in this chapter; however, still quite an intriguing paper in its own right.*

[G74] "Survey of Virtual Machine Research"
R.P. Goldberg
IEEE Computer, Volume 7, Number 6
A terrific survey of a lot of old virtual machine research.

[SVL01] "Virtualizing I/O Devices on VMware Workstation's
Hosted Virtual Machine Monitor"
Jeremy Sugerman, Ganesh Venkitachalam and Beng-Hong Lim
USENIX '01, Boston, Massachusetts
Provides a good overview of how I/O works in VMware using a hosted architecture which exploits many native OS features to avoid reimplementing them within the VMM.

[V98] VMware corporation.
Available: http://www.vmware.com/
This may be the most useless reference in this book, as you can clearly look this up yourself. Anyhow, the company was founded in 1998 and is a leader in the field of virtualization.

[S+03] "Semantically-Smart Disk Systems"
Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
FAST '03, San Francisco, California, March 2003
Our work again, this time showing how a dumb block-based device can infer much about what the file system above it is doing, such as deleting a file. The technology used therein enables interesting new functionality within a block device, such as secure delete, or more reliable storage.

[WSG02] "Scale and Performance in the Denali Isolation Kernel"
Andrew Whitaker, Marianne Shaw, and Steven D. Gribble
OSDI '02, Boston, Massachusetts
The paper that introduces the term para-virtualization. Although one can argue that Bugnion et al. [B+97] introduce the idea of para-virtualization in the Disco paper, Whitaker et al. take it further and show how the idea can be more general than what was thought before.

C

A Dialogue on Monitors

Professor: *So it's you again, huh?*

Student: *I bet you are getting quite tired by now, being so, well you know, old? Not that 50 years old is that old, really.*

Professor: *I'm not 50! I've just turned 40, actually. But goodness, I guess to you, being 20-something ...*

Student: *... 19, actually ...*

Professor: *(ugh) ... yes, 19, whatever, I guess 40 and 50 seem kind of similar. But trust me, they're not. At least, that's what my 50-year old friends tell me.*

Student: *Anyhow ...*

Professor: *Ah yes! What are we talking about again?*

Student: *Monitors. Not that I know what a* ***monitor*** *is, except for some kind of old-fashioned name for the computer display sitting in front of me.*

Professor: *Yes, this is a whole different type of thing. It's an old concurrency primitive, designed as a way to incorporate locking automatically into object-oriented programs.*

Student: *Why not include it in the section on concurrency then?*

Professor: *Well, most of the book is about C programming and the POSIX threads libraries, where there are no monitors, so there's that. But there are some historical reasons to at least include the information on the topic, so here it is, I guess.*

Student: *Ah, history. That's for old people, like you, right?*

Professor: *(glares)*

Student: *Oh take it easy. I kid!*

Professor: *I can't wait until you take the final exam...*

D

Monitors (Deprecated)

Around the time concurrent programming was becoming a big deal, object-oriented programming was also gaining ground. Not surprisingly, people started to think about ways to merge synchronization into a more structured programming environment.

One such approach that emerged was the **monitor**. First described by Per Brinch Hansen [BH73] and later refined by Tony Hoare [H74], the idea behind a monitor is quite simple. Consider the following pretend monitor written in C++ notation:

```
monitor class account {
private:
    int balance = 0;

public:
    void deposit(int amount) {
        balance = balance + amount;
    }
    void withdraw(int amount) {
        balance = balance - amount;
    }
};
```

Figure D.1: **A Pretend Monitor Class**

Note: this is a "pretend" class because C++ does not support monitors, and hence the **monitor** keyword does not exist. However, Java does support monitors, with what are called **synchronized** methods. Below, we will examine both how to make something quite like a monitor in C/C++, as well as how to use Java synchronized methods.

In this example, you may notice we have our old friend the account and some routines to deposit and withdraw an amount from the balance. As you also may notice, these are **critical sections**; if they are called by multiple threads concurrently, you have a race condition and the potential for an incorrect outcome.

In a monitor class, you don't get into trouble, though, because the monitor guarantees that **only one thread can be active within the monitor at a time**. Thus, our above example is a perfectly safe and working

piece of code; multiple threads can call deposit() or withdraw() and know that mutual exclusion is preserved.

How does the monitor do this? Simple: with a lock. Whenever a thread tries to call a monitor routine, it implicitly tries to acquire the monitor lock. If it succeeds, then it will be able to call into the routine and run the method's code. If it does not, it will block until the thread that is in the monitor finishes what it is doing. Thus, if we wrote a C++ class that looked like the following, it would accomplish the exact same goal as the monitor class above:

```
class account {
private:
    int balance = 0;
    pthread_mutex_t monitor;

public:
    void deposit(int amount) {
        pthread_mutex_lock(&monitor);
        balance = balance + amount;
        pthread_mutex_unlock(&monitor);
    }
    void withdraw(int amount) {
        pthread_mutex_lock(&monitor);
        balance = balance - amount;
        pthread_mutex_unlock(&monitor);
    }
};
```

Figure D.2: **A C++ Class that acts like a Monitor**

Thus, as you can see from this example, the monitor isn't doing too much for you automatically. Basically, it is just acquiring a lock and releasing it. By doing so, we achieve what the monitor requires: only one thread will be active within deposit() or withdraw(), as desired.

D.1 Why Bother with Monitors?

You might wonder why monitors were invented at all, instead of just using explicit locking. At the time, object-oriented programming was just coming into fashion. Thus, the idea was to gracefully blend some of the key concepts in concurrent programming with some of the basic approaches of object orientation. Nothing more than that.

D.2 Do We Get More Than Automatic Locking?

Back to business. As we know from our discussion of semaphores, just having locks is not quite enough; for example, to implement the producer/consumer solution, we previously used semaphores to both put threads to sleep when waiting for a condition to change (e.g., a producer waiting for a buffer to be emptied), as well as to wake up a thread when a particular condition has changed (e.g., a consumer signaling that it has indeed emptied a buffer).

```
monitor class BoundedBuffer {
  private:
    int buffer[MAX];
    int fill, use;
    int fullEntries = 0;
    cond_t empty;
    cond_t full;

  public:
    void produce(int element) {
        if (fullEntries == MAX)     // line P0
            wait(&empty);           // line P1
        buffer[fill] = element;     // line P2
        fill = (fill + 1) % MAX;    // line P3
        fullEntries++;              // line P4
        signal(&full);              // line P5
    }

    int consume() {
        if (fullEntries == 0)       // line C0
            wait(&full);            // line C1
        int tmp = buffer[use];      // line C2
        use = (use + 1) % MAX;      // line C3
        fullEntries--;              // line C4
        signal(&empty);             // line C5
        return tmp;                 // line C6
    }
}
```

Figure D.3: **Producer/Consumer with Monitors and Hoare Semantics**

Monitors support such functionality through an explicit construct known as a **condition variable**. Let's take a look at the producer/consumer solution, here written with monitors and condition variables.

In this monitor class, we have two routines, produce() and consume(). A producer thread would repeatedly call produce() to put data into the bounded buffer, while a consumer() would repeatedly call consume(). The example is a modern paraphrase of Hoare's solution [H74].

You should notice some similarities between this code and the semaphore-based solution in the previous note. One major difference is how condition variables must be used in concert with an explicit **state variable**; in this case, the integer `fullEntries` determines whether a producer or consumer must wait, depending on its state. Semaphores, in contrast, have an internal numeric value which serves this same purpose. Thus, condition variables must be paired with some kind of external state value in order to achieve the same end.

The most important aspect of this code, however, is the use of the two condition variables, empty and full, and the respective `wait()` and `signal()` calls that employ them. These operations do exactly what you might think: wait() blocks the calling thread on a given condition; `signal()` wakes one waiting thread that is waiting on the condition.

However, there are some subtleties in how these calls operate; understanding the semantics of these calls is critically important to understand-

ing why this code works. In what researchers in operating systems call **Hoare semantics** (yes, a somewhat unfortunate name), the `signal()` immediately wakes one waiting thread and runs it; thus, the monitor lock, which is implicitly held by the running thread, immediately is transferred to the woken thread which then runs until it either blocks or exits the monitor. Note that there may be more than one thread waiting; `signal()` only wakes one waiting thread and runs it, while the others must wait for a subsequent signal.

A simple example will help us understand this code better. Imagine there are two threads, one a producer and the other a consumer. The consumer gets to run first, and calls `consume()`, only to find that fullEntries = 0 `(C0)`, as there is nothing in the buffer yet. Thus, it calls `wait(&full)` `(C1)`, and waits for a buffer to be filled. The producer then runs, finds it doesn't have to wait `(P0)`, puts an element into the buffer `(P2)`, increments the fill index `(P3)` and the fullEntries count `(P4)`, and calls `signal(&full)` `(P5)`. In Hoare semantics, the producer does not continue running after the signal; rather, the signal immediately transfers control to the waiting consumer, which returns from `wait()` `(C1)` and immediately consumes the element produced by the producer `(C2)` and so on. Only after the consumer returns will the producer get to run again and return from the produce() routine.

D.3 Where Theory Meets Practice

Tony Hoare, who wrote the solution above and came up with the exact semantics for `signal()` and `wait()`, was a theoretician. Clearly a smart guy, too; he came up with quicksort after all [H61]. However, the semantics of signaling and waiting, as it turns out, were not ideal for a real implementation. As the old saying goes, in theory, there is no difference between theory and practice, but in practice, there is.

OLD SAYING: THEORY VS. PRACTICE

The old saying is "in theory, there is no difference between theory and practice, but in practice, there is." Of course, only practitioners tell you this; a theory person could undoubtedly prove that it is not true.

A few years later, Butler Lampson and David Redell of Xerox PARC were building a concurrent language known as **Mesa**, and decided to use monitors as their basic concurrency primitive [LR80]. They were well-known systems researchers, and they soon found that Hoare semantics, while more amenable to proofs, were hard to realize in a real system (there are a lot of reasons for this, perhaps too many to go through here).

In particular, to build a working monitor implementation, Lampson and Redell decided to change the meaning of signal() in a subtle but critical way. The signal() routine now was just considered a **hint** [L83]; it

would move a single waiting thread from the blocked state to a runnable state, but it would not run it immediately. Rather, the signaling thread would retain control until it exited the monitor and was descheduled.

D.4 Oh Oh, A Race

Given these new **Mesa semantics**, let us again reexamine the code above. Imagine again a consumer (consumer 1) who enters the monitor and finds the buffer empty and thus waits `(C1)`. Now the producer comes along and fills the buffer and signals that a buffer has been filled, moving the waiting consumer from blocked on the full condition variable to ready. The producer keeps running for a while, and eventually gives up the CPU.

But Houston, we have a problem. Can you see it? Imagine a different consumer (consumer 2) now calls into the consume() routine; it will find a full buffer, consume it, and return, setting fullEntries to 0 in the meanwhile. Can you see the problem yet? Well, here it comes. Our old friend consumer 1 now finally gets to run, and returns from wait(), expecting a buffer to be full `(C1...)`; unfortunately, this is no longer true, as consumer 2 snuck in and consumed the buffer before consumer 1 had a chance to consume it. Thus, the code doesn't work, because in the time between the signal() by the producer and the return from wait() by consumer 1, the condition has changed. This timeline illustrates the problem:

```
      Producer                Consumer1                Consumer2
                        C0 (fullEntries=0)
                        C1 (Consumer 1: blocked)
P0 (fullEntries=0)
P2
P3
P4 (fullEntries=1)
P5 (Consumer1: ready)
                                                 C0 (fullEntries=1)
                                                 C2
                                                 C3
                                                 C4 (fullEntries=0)
                                                 C5
                                                 C6
                        C2 (using a buffer,
                            fullEntries=0!)
```

Figure D.4: **Why the Code doesn't work with Hoare Semantics**

Fortunately, the switch from Hoare semantics to Mesa semantics requires only a small change by the programmer to realize a working solution. Specifically, when woken, a thread should *recheck* the condition it was waiting on; because signal() is only a hint, it is possible that the condition has changed (even multiple times) and thus may not be in the desired state when the waiting thread runs. In our example, two lines of code must change, lines P0 and C0:

```
public:
  void produce(int element) {
      while (fullEntries == MAX) // line P0 (CHANGED IF->WHILE)
          wait(&empty);          // line P1
      buffer[fill] = element;    // line P2
      fill = (fill + 1) % MAX;   // line P3
      fullEntries++;             // line P4
      signal(&full);             // line P5
  }

  int consume() {
      while (fullEntries == 0)   // line C0 (CHANGED IF->WHILE)
          wait(&full);           // line C1
      int tmp = buffer[use];     // line C2
      use = (use + 1) % MAX;     // line C3
      fullEntries--;             // line C4
      signal(&empty);            // line C5
      return tmp;                // line C6
  }
```

Figure D.5: **Producer/Consumer with Monitors and Mesa Semantics**

Not too hard after all. Because of the ease of this implementation, virtually any system today that uses condition variables with signaling and waiting uses Mesa semantics. Thus, if you remember nothing else at all from this class, you can just remember: **always recheck the condition after being woken!** Put in even simpler terms, **use while loops** and not if statements when checking conditions. Note that this is always correct, even if somehow you are running on a system with Hoare semantics; in that case, you would just needlessly retest the condition an extra time.

D.5 Peeking Under The Hood A Bit

To understand a bit better why Mesa semantics are easier to implement, let's understand a little more about the implementation of Mesa monitors. In their work [LR80], Lampson and Redell describe three different types of queues that a thread can be a part of at a given time: the **ready** queue, a **monitor lock** queue, and a **condition variable** queue. Note that a program might have multiple monitor classes and multiple condition variable instances; there is a queue per instance of said items.

With a single bounded buffer monitor, we thus have four queues to consider: the ready queue, a single monitor queue, and two condition variable queues (one for the full condition and one for the empty). To better understand how a thread library manages these queues, what we will do is show how a thread transitions through these queues in the producer/consumer example.

In this example, we walk through a case where a consumer might be woken up but find that there is nothing to consume. Let us consider the following timeline. On the left are two consumers (Con1 and Con2) and a producer (Prod) and which line of code they are executing; on the right is the state of each of the four queues we are following for this example:

```
 t | Con1 Con2 Prod | Mon | Empty | Full | FE | Comment
-------------------------------------------------------------------------
 0  C0                                      0
 1  C1                                Con1  0   Con1 waiting on full
 2 <Context switch>                   Con1  0   switch: Con1 to Prod
 3            P0                      Con1  0
 4            P2                      Con1  0   Prod doesn't wait (FE=0)
 5            P3                      Con1  0
 6            P4                      Con1  1   Prod updates fullEntries
 7            P5                            1   Prod signals: Con1 now ready
 8 <Context switch>                         1   switch: Prod to Con2
 9        C0                                1   switch to Con2
10        C2                                1   Con2 doesn't wait (FE=1)
11        C3                                1
12        C4                                0   Con2 changes fullEntries
13        C5                                0   Con2 signals empty (no waiter)
14        C6                                0   Con2 done
15 <Context switch>                         0   switch: Con2 to Con1
16   C0                                     0   recheck fullEntries: 0!
17   C1                              Con1   0   wait on full again
```

Figure D.6: **Tracing Queues during a Producer/Consumer Run**

the ready queue of runnable processes, the monitor lock queue called Monitor, and the empty and full condition variable queues. We also track time (t), the thread that is running (square brackets around the thread on the ready queue that is running), and the value of fullEntries (FE).

As you can see from the timeline, consumer 2 (Con2) sneaks in and consumes the available data (t=9..14) before consumer 1 (Con1), who was waiting on the full condition to be signaled (since t=1), gets a chance to do so. However, Con1 does get woken by the producer's signal (t=7), and thus runs again even though the buffer is empty by the time it does so. If Con1 didn't recheck the state variable fullEntries (t=16), it would have erroneously tried to consume data when no data was present to consume. Thus, this natural implementation is exactly what leads us to Mesa semantics (and not Hoare).

D.6 Other Uses Of Monitors

In their paper on Mesa, Lampson and Redell also point out a few places where a different kind of signaling is needed. For example, consider the following memory allocator (Figure D.7).

Many details are left out of this example, in order to allow us to focus on the conditions for waking and signaling. It turns out the signal/wait code above does not quite work; can you see why?

Imagine two threads call allocate. The first calls allocate(20) and the second allocate(10). No memory is available, and thus both threads call wait() and block. Some time later, a different thread comes along and calls free(p, 15), and thus frees up 15 bytes of memory. It then signals that it has done so. Unfortunately, it wakes the thread waiting for 20 bytes; that thread rechecks the condition, finds that only 15 bytes are available, and

```
monitor class allocator {
    int available; // how much memory is available?
    cond_t c;

    void *allocate(int size) {
        while (size > available)
            wait(&c);
        available -= size;
        // and then do whatever the allocator should do
        // and return a chunk of memory
    }

    void free(void *pointer, int size) {
        // free up some memory
        available += size;
        signal(&c);
    }
};
```

Figure D.7: **A Simple Memory Allocator**

calls wait() again. The thread that could have benefited from the free of 15 bytes, i.e., the thread that called allocate(10), is not woken.

Lampson and Redell suggest a simple solution to this problem. Instead of a signal() which wakes a single waiting thread, they employ a **broadcast()** which wakes *all* waiting threads. Thus, all threads are woken up, and in the example above, the thread waiting for 10 bytes will find 15 available and succeed in its allocation.

In Mesa semantics, using a broadcast() is *always* correct, as all threads should recheck the condition of interest upon waking anyhow. However, it may be a performance problem, and thus should only be used when needed. In this example, a broadcast() might wake hundreds of waiting threads, only to have one successfully continue while the rest immediately block again; this problem, sometimes known as a **thundering herd**, is costly, due to all the extra context switches that occur.

D.7 Using Monitors To Implement Semaphores

You can probably see a lot of similarities between monitors and semaphores. Not surprisingly, you can use one to implement the other. Here, we show how you might implement a semaphore class using a monitor (Figure D.8).

As you can see, wait() simply waits for the value of the semaphore to be greater than 0, and then decrements its value, whereas post() increments the value and wakes one waiting thread (if there is one). It's as simple as that.

To use this class as a binary semaphore (i.e., a lock), you just initialize the semaphore to 1, and then put wait()/post() pairs around critical sections. And thus we have shown that monitors can be used to implement semaphores.

```
monitor class Semaphore {
    int s; // value of the semaphore
    Semaphore(int value) {
        s = value;
    }
    void wait() {
        while (s <= 0)
            wait();
        s--;
    }
    void post() {
        s++;
        signal();
    }
};
```

Figure D.8: **Implementing a Semaphore with a Monitor**

D.8 Monitors in the Real World

We already mentioned above that we were using "pretend" monitors; C++ has no such concept. We now show how to make a monitor-like C++ class, and how Java uses synchronized methods to achieve a similar end.

A C++ Monitor of Sorts

Here is the producer/consumer code written in C++ with locks and condition variables (Figure D.9). You can see in this code example that there is little difference between the pretend monitor code and the working C++ class we have above. Of course, one obvious difference is the explicit use of a lock "monitor". More subtle is the switch to the POSIX standard `pthread_cond_signal()` and `pthread_cond_wait()` calls. In particular, notice that when calling `pthread_cond_wait()`, one also passes in the lock that is held at the time of waiting. The lock is needed inside `pthread_cond_wait()` because it must be released when this thread is put to sleep and re-acquired before it returns to the caller (the same behavior as within a monitor but again with explicit locks).

A Java Monitor

Interestingly, the designers of Java decided to use monitors as they thought they were a graceful way to add synchronization primitives into a language. To use them, you just use add the keyword **synchronized** to the method or set of methods that you wish to use as a monitor (here is an example from Sun's own documentation site [S12a,S12b]):

This code does exactly what you think it should: provide a counter that is thread safe. Because only one thread is allowed into the monitor at a time, only one thread can update the value of "c", and thus a race condition is averted.

```
class BoundedBuffer {
  private:
    int buffer[MAX];
    int fill, use;
    int fullEntries;
    pthread_mutex_t monitor; // monitor lock
    pthread_cond_t empty;
    pthread_cond_t full;

  public:
    BoundedBuffer() {
        use = fill = fullEntries = 0;
    }
    void produce(int element) {
        pthread_mutex_lock(&monitor);
        while (fullEntries == MAX)
            pthread_cond_wait(&empty, &monitor);
        buffer[fill] = element;
        fill = (fill + 1) % MAX;
        fullEntries++;
        pthread_cond_signal(&full);
        pthread_mutex_unlock(&monitor);
    }

    int consume() {
        pthread_mutex_lock(&monitor);
        while (fullEntries == 0)
            pthread_cond_wait(&full, &monitor);
        int tmp = buffer[use];
        use = (use + 1) % MAX;
        fullEntries--;
        pthread_cond_signal(&empty);
        pthread_mutex_unlock(&monitor);
        return tmp;
    }
}
```

Figure D.9: **C++ Producer/Consumer with a "Monitor"**

Java and the Single Condition Variable

In the original version of Java, a condition variable was also supplied with each synchronized class. To use it, you would call either **wait()** or **notify()** (sometimes the term notify is used instead of signal, but they mean the same thing). Oddly enough, in this original implementation, there was no way to have two (or more) condition variables. You may have noticed in the producer/consumer solution, we always use two: one for signaling a buffer has been emptied, and another for signaling that a buffer has been filled.

To understand the limitations of only providing a single condition variable, let's imagine the producer/consumer solution with only a single condition variable. Imagine two consumers run first, and both get stuck waiting. Then, a producer runs, fills a single buffer, wakes a single

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
```

Figure D.10: **A Simple Java Class with Synchronized Methods**

consumer, and then tries to fill again but finds the buffer full (MAX=1). Thus, we have a producer waiting for an empty buffer, a consumer waiting for a full buffer, and a consumer who had been waiting about to run because it has been woken.

The consumer then runs and consumes the buffer. When it calls notify(), though, it wakes a single thread that is waiting on the condition. Because there is only a single condition variable, the consumer might wake the waiting **consumer,** instead of the waiting producer. Thus, the solution does not work.

To remedy this problem, one can again use the broadcast solution. In Java, one calls **notifyAll()** to wake all waiting threads. In this case, the consumer would wake a producer and a consumer, but the consumer would find that fullEntries is equal to 0 and go back to sleep, while the producer would continue. As usual, waking all waiters can lead to the thundering herd problem.

Because of this deficiency, Java later added an explicit Condition class, thus allowing for a more efficient solution to this and other similar concurrency problems.

D.9 Summary

We have seen the introduction of monitors, a structuring concept developed by Brinch Hansen and and subsequently Hoare in the early seventies. When running inside the monitor, a thread implicitly holds a monitor lock, and thus prevents other threads from entering the monitor, allowing the ready construction of mutual exclusion.

We also have seen the introduction of explicit condition variables, which allow threads to signal() and wait() much like we saw with semaphores in the previous note. The semantics of signal() and wait() are critical; because all modern systems implement **Mesa** semantics, a recheck of the condition that the thread went to sleep on is required for correct execution. Thus, signal() is just a **hint** that something has changed; it is the responsibility of the woken thread to make sure the conditions are right

for its continued execution.

Finally, because C++ has no monitor support, we saw how to emulate monitors with explicit pthread locks and condition variables. We also saw how Java supports monitors with its synchronized routines, and some of the limitations of only providing a single condition variable in such an environment.

References

[BH73] "Operating System Principles"
Per Brinch Hansen, Prentice-Hall, 1973
Available: http://portal.acm.org/citation.cfm?id=540365
One of the first books on operating systems; certainly ahead of its time. Introduced monitors as a concurrency primitive.

[H74] "Monitors: An Operating System Structuring Concept"
C.A.R. Hoare
CACM, Volume 17:10, pages 549–557, October 1974
An early reference to monitors; however, Brinch Hansen probably was the true inventor.

[H61] "Quicksort: Algorithm 64"
C.A.R. Hoare
CACM, Volume 4:7, July 1961
The famous quicksort algorithm.

[LR80] "Experience with Processes and Monitors in Mesa"
B.W. Lampson and D.R. Redell
CACM, Volume 23:2, pages 105–117, February 1980
An early and important paper highlighting the differences between theory and practice.

[L83] "Hints for Computer Systems Design"
Butler Lampson
ACM Operating Systems Review, 15:5, October 1983
Lampson, a famous systems researcher, loved using hints in the design of computer systems. A hint is something that is often correct but can be wrong; in this use, a signal() is telling a waiting thread that it changed the condition that the waiter was waiting on, but not to trust that the condition will be in the desired state when the waiting thread wakes up. In this paper about hints for designing systems, one of Lampson's general hints is that you should use hints. It is not as confusing as it sounds.

[S12a] "Synchronized Methods"
Sun documentation
http://java.sun.com/docs/books/tutorial/essential/concurrency/syncmeth.html

[S12b] "Condition Interface"
Sun documentation
http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/locks/Condition.html

E

A Dialogue on Labs

Student: *Is this our final dialogue?*

Professor: *I hope so! You've been becoming quite a pain, you know!*

Student: *Yes, I've enjoyed our conversations too. What's up here?*

Professor: *It's about the projects you should be doing as you learn this material; you know, actual programming, where you do some real work instead of this incessant talking and reading. The real way to learn!*

Student: *Sounds important. Why didn't you tell me earlier?*

Professor: *Well, hopefully those using this book actually do look at this part earlier, all throughout the course. If not, they're really missing something.*

Student: *Seems like it. So what are the projects like?*

Professor: *Well, there are two types of projects. The first set are what you might call **systems programming** projects, done on machines running Linux and in the C programming environment. This type of programming is quite useful to know, as when you go off into the real world, you very well might have to do some of this type of hacking yourself.*

Student: *What's the second type of project?*

Professor: *The second type is based inside a real kernel, a cool little teaching kernel developed at MIT called **xv6**. It is a "port" of an old version of* UNIX *to Intel x86, and is quite neat! With these projects, instead of writing code that interacts with the kernel (as you do in systems programming), you actually get to re-write parts of the kernel itself!*

Student: *Sounds fun! So what should we do in a semester? You know, there are only so many hours in the day, and as you professors seem to forget, we students take four or five courses, not just yours!*

Professor: *Well, there is a lot of flexibility here. Some classes just do all systems programming, because it is so practical. Some classes do all xv6 hacking, because it really gets you to see how operating systems work. And some, as you may have guessed, do a mix, starting with some systems programming, and then doing xv6 at the end. It's really up to the professor of a particular class.*

Student: *(sighing) Professors have all the control, it seems...*

Professor: *Oh, hardly! But that little control they do get to exercise is one of the fun parts of the job. Deciding on assignments is important you know — and not something any professor takes lightly.*

Student: *Well, that is good to hear. I guess we should see what these projects are all about...*

Professor: *OK. And one more thing: if you're interested in the systems programming part, there is also a little tutorial about the* UNIX *and C programming environment.*

Student: *Sounds almost too useful to be true.*

Professor: *Well, take a look. You know, classes are supposed to be about useful things, sometimes!*

F

Laboratory: Tutorial

This is a very brief document to familiarize you with the basics of the C programming environment on UNIX systems. It is not comprehensive or particularly detailed, but should just give you enough to get you going.

A couple of general points of advice about programming: if you want to become an expert programmer, you need to master more than just the syntax of a language. Specifically, you should **know your tools**, **know your libraries**, and **know your documentation**. The tools that are relevant to C compilation are **gcc**, **gdb**, and maybe **ld**. There are tons of library routines that are also available to you, but fortunately a lot of functionality is included in **libc**, which is linked with all C programs by default — all you need to do is include the right header files. Finally, knowing how to find the library routines you need (e.g., learning to find and read man pages) is a skill worth acquiring. We'll talk about each of these in more detail later on.

Like (almost) everything worth doing in life, becoming an expert in these domains takes time. Spending the time up-front to learn more about the tools and environment is definitely well worth the effort.

F.1 A Simple C Program

We'll start with a simple C program, perhaps saved in the file "hw.c". Unlike Java, there is not necessarily a connection between the file name and the contents of the file; thus, use your common sense in naming files in a manner that is appropriate.

The first line specifies a file to include, in this case `stdio.h`, which "prototypes" many of the commonly used input/output routines; the one we are interested in is `printf()`. When you use the `#include` directive, you are telling the C preprocessor (`cpp`) to find a particular file (e.g., `stdio.h`) and to insert it directly into your code at the spot of the `#include`. By default, `cpp` will look in the directory `/usr/include/` to try to find the file.

The next part specifies the signature of the `main()` routine, namely that it returns an integer (`int`), and will be called with two arguments,

```
/* header files go up here */
/* note that C comments are enclosed within a slash and a star, and
   may wrap over lines */
// if you use gcc, two slashes will work too (and may be preferred)
#include <stdio.h>

/* main returns an integer */
int main(int argc, char *argv[]) {
    /* printf is our output function;
       by default, writes to standard out */
    /* printf returns an integer, but we ignore that */
    printf("hello, world\n");

    /* return 0 to indicate all went well */
    return(0);
}
```

an integer `argc`, which is a count of the number of arguments on the command line, and an array of pointers to characters (`argv`), each of which contain a word from the command line, and the last of which is null. There will be more on pointers and arrays below.

The program then simply prints the string "hello, world" and advances the output stream to the next line, courtesy of the backslash followed by an "n" at the end of the call to `printf()`. Afterwards, the program completes by returning a value, which is passed back to the shell that executed the program. A script or the user at the terminal could check this value (in csh and tcsh shells, it is stored in the `status` variable), to see whether the program exited cleanly or with an error.

F.2 Compilation and Execution

We'll now learn how to compile the program. Note that we will use `gcc` as our example, though on some platforms you may be able to use a different (native) compiler, `cc`.

At the shell prompt, you just type:

```
prompt> gcc hw.c
```

`gcc` is not really the compiler, but rather the program called a "compiler driver"; thus it coordinates the many steps of the compilation. Usually there are four to five steps. First, `gcc` will execute `cpp`, the C preprocessor, to process certain directives (such as `#define` and `#include`. The program `cpp` is just a source-to-source translator, so its end-product is still just source code (i.e., a C file). Then the real compilation will begin, usually a command called `cc1`. This will transform source-level C code into low-level assembly code, specific to the host machine. The assembler `as` will then be executed, generating object code (bits and things that machines can really understand), and finally the link-editor (or linker) `ld` will put it all together into a final executable program. Fortunately(!), for most purposes, you can blithely be unaware of how `gcc` works, and just use it with the proper flags.

The result of your compilation above is an executable, named (by default) `a.out`. To then run the program, we simply type:

```
prompt> ./a.out
```

When we run this program, the OS will set argc and argv properly so that the program can process the command-line arguments as need be. Specifically, `argc` will be equal to `1`, `argv[0]` will be the string "./a.out", and `argv[1]` will be null, indicating the end of the array.

F.3 Useful Flags

Before moving on to the C language, we'll first point out some useful compilation flags for `gcc`.

```
prompt> gcc -o hw hw.c  # -o: to specify the executable name
prompt> gcc -Wall hw.c  # -Wall: gives much better warnings
prompt> gcc -g hw.c     # -g: to enable debugging with gdb
prompt> gcc -O hw.c     # -O: to turn on optimization
```

Of course, you may combine these flags as you see fit (e.g., `gcc -o hw -g -Wall hw.c`). Of these flags, you should always use `-Wall`, which gives you lots of extra warnings about possible mistakes. **Don't ignore the warnings!** Instead, fix them and thus make them blissfully disappear.

F.4 Linking with Libraries

Sometimes, you may want to use a library routine in your program. Because so many routines are available in the C library (which is automatically linked with every program), all you usually have to do is find the right `#include` file. The best way to do that is via the **manual pages**, usually just called the **man pages**.

For example, let's say you want to use the `fork()` system call[1]. By typing `man fork` at the shell prompt, you will get back a text description of how `fork()` works. At the very top will be a short code snippet, and that will tell you which files you need to `#include` in your program in order to get it to compile. In the case of `fork()`, you need to `#include` both `sys/types.h` and `unistd.h`, which would be accomplished as follows:

```
#include <sys/types.h>
#include <unistd.h>
```

[1] Note that `fork()` is a system call, and not just a library routine. However, the C library provides C wrappers for all the system calls, each of which simply trap into the operating system.

However, some library routines do not reside in the C library, and therefore you will have to do a little more work. For example, the math library has many useful routines, such as sines, cosines, tangents, and the like. If you want to include the routine `tan()` in our code, you should again first check the man page. At the top of the Linux man page for tan, you will see the following two lines:

```
#include <math.h>
...
Link with -lm.
```

The first line you already should understand — you need to `#include` the math library, which is found in the standard location in the file system (i.e., `/usr/include/math.h`). However, what the next line is telling you is how to "link" your program with the math library. A number of useful libraries exist and can be linked with; many of those reside in `/usr/lib`; it is indeed where the math library is found.

There are two types of libraries: statically-linked libraries (which end in `.a`), and dynamically-linked ones (which end in `.so`). Statically-linked libraries are combined directly into your executable; that is, the low-level code for the library is inserted into your executable by the linker, and results in a much larger binary object. Dynamic linking improves on this by just including the reference to a library in your program executable; when the program is run, the operating system loader dynamically links in the library. This method is preferred over the static approach because it saves disk space (no unnecessarily large executables are made) and allows applications to share library code and static data in memory. In the case of the math library, both static and dynamic versions are available, with the static version called `/usr/lib/libm.a` and the dynamic one `/usr/lib/libm.so`.

In any case, to link with the math library, you need to specify the library to the link-editor; this can be achieved by invoking `gcc` with the right flags.

```
prompt> gcc -o hw hw.c -Wall -lm
```

The `-lXXX` flag tells the linker to look for `libXXX.so` or `libXXX.a`, probably in that order. If for some reason you insist on the static library over the dynamic one, there is another flag you can use — see you if you can find out what it is. People sometimes prefer the static version of a library because of the slight performance cost associated with using dynamic libraries.

One final note: if you want the compiler to search for headers in a different path than the usual places, or want it to link with libraries that you specify, you can use the compiler flag `-I/foo/bar` to look for headers in the directory `/foo/bar`, and the `-L/foo/bar` flag to look for libraries in the `/foo/bar` directory. One common directory to specify in this manner is "`.`" (called "dot"), which is UNIX shorthand for the current directory.

Note that the `-I` flag should go on a compile line, and the `-L` flag on the link line.

F.5 Separate Compilation

Once a program starts to get large enough, you may want to split it into separate files, compiling each separately, and then link them together. For example, say you have two files, `hw.c` and `helper.c`, and you wish to compile them individually, and then link them together.

```
# we are using -Wall for warnings, -O for optimization
prompt> gcc -Wall -O -c hw.c
prompt> gcc -Wall -O -c helper.c
prompt> gcc -o hw hw.o helper.o -lm
```

The `-c` flag tells the compiler just to produce an object file — in this case, files called `hw.o` and `helper.o`. These files are not executables, but just machine-level representations of the code within each source file. To combine the object files into an executable, you have to "link" them together; this is accomplished with the third line `gcc -o hw hw.o helper.o`). In this case, `gcc` sees that the input files specified are not source files (`.c`), but instead are object files (`.o`), and therefore skips right to the last step and invoked the link-editor `ld` to link them together into a single executable. Because of its function, this line is often called the "link line", and would be where you specify link-specific commands such as `-lm`. Analogously, flags such as `-Wall` and `-O` are only needed in the compile phase, and therefore need not be included on the link line but rather only on compile lines.

Of course, you could just specify all the C source files on a single line to `gcc` (`gcc -Wall -O -o hw hw.c helper.c`), but this requires the system to recompile every source-code file, which can be a time-consuming process. By compiling each individually, you can save time by only recompiling those files that have changed during your editing, and thus increase your productivity. This process is best managed by another program, `make`, which we now describe.

F.6 Makefiles

The program `make` lets you automate much of your build process, and is thus a crucially important tool for any serious program (and programmer). Let's take a look at a simple example, saved in a file called `Makefile`.

To build your program, now all you have to do is type:

```
prompt> make
```

This will (by default) look for `Makefile` or `makefile`, and use that as its input (you can specify a different makefile with a flag; read the

```
hw: hw.o helper.o
    gcc -o hw hw.o helper.o -lm

hw.o: hw.c
    gcc -O -Wall -c hw.c

helper.o: helper.c
    gcc -O -Wall -c helper.c

clean:
    rm -f hw.o helper.o hw
```

man pages to find out which). The gnu version of `make`, `gmake`, is more fully featured than traditional make, so we will focus upon it for the rest of this discussion (though we will use the two terms interchangeably). Most of these notes are based on the `gmake` info page; to see how to find those pages, see the Documentation section below. Also note: on Linux systems, `gmake` and `make` are one and the same.

Makefiles are based on rules, which are used to decide what needs to happen. The general form of a rule:

```
target: prerequisite1 prerequisite2 ...
    command1
    command2
    ...
```

A **target** is usually the name of a file that is generated by a program; examples of targets are executable or object files. A target can also be the name of an action to carry out, such as "clean" in our example.

A **prerequisite** is a file that is used as input to create the target. A target often depends on several files. For example, to build the executable `hw`, we need two object files to be built first: `hw.o` and `helper.o`.

Finally, a **command** is an action that `make` carries out. A rule may have more than one command, each on its own line. **Important:** You have to put a single tab character at the beginning of every command line! If you just put spaces, `make` will print out some obscure error message and exit.

Usually a command is in a rule with prerequisites and serves to create a target file if any of the prerequisites change. However, the rule that specifies commands for the target need not have prerequisites. For example, the rule containing the delete command associated with the target "clean" does not have prerequisites.

Going back to our example, when make is executed, it roughly works like this: First, it comes to the target `hw`, and it realizes that to build it, it must have two prerequisites, `hw.o` and `helper.o`. Thus, `hw` depends on those two object files. Make then will examine each of those targets. In examining `hw.o`, it will see that it depends on `hw.c`. Here is the key: if `hw.c` has been modified more recently than `hw.o` has been created, `make` will know that `hw.o` is out of date and should be generated anew; in that case, it will execute the command line, `gcc -O -Wall -c hw.c`, which generates `hw.o`. Thus, if you are compiling a large program, `make` will know which object files need to be re-generated based on their depen-

dencies, and will only do the necessary amount of work to recreate the executable. Also note that `hw.o` will be created in the case that it does not exist at all.

Continuing along, `helper.o` may also be regenerated or created, based on the same criteria as defined above. When both of the object files have been created, make is now ready to execute the command to create the final executable, and goes back and does so: `gcc -o hw hw.o helper.o -lm`.

Up until now, we've been ignoring the `clean` target in the makefile. To use it, you have to ask for it explicitly. Type

```
prompt> make clean
```

This will execute the command on the command line. Because there are no prerequisites for the `clean` target, typing `make clean` will always result in the command(s) being executed. In this case, the `clean` target is used to remove the object files and executable, quite handy if you wish to rebuild the entire program from scratch.

Now you might be thinking, "well, this seems OK, but these makefiles sure are cumbersome!" And you'd be right — if they always had to be written like this. Fortunately, there are a lot of shortcuts that make `make` even easier to use. For example, this makefile has the same functionality but is a little nicer to use:

```
# specify all source files here
SRCS = hw.c helper.c
# specify target here (name of executable)
TARG = hw
# specify compiler, compile flags, and needed libs
CC   = gcc
OPTS = -Wall -O
LIBS = -lm

# this translates .c files in src list to .o's
OBJS = $(SRCS:.c=.o)

# all is not really needed, but is used to generate the target
all: $(TARG)

# this generates the target executable
$(TARG): $(OBJS)
    $(CC) -o $(TARG) $(OBJS) $(LIBS)

# this is a generic rule for .o files
%.o: %.c
    $(CC) $(OPTS) -c $< -o $@

# and finally, a clean line
clean:
    rm -f $(OBJS) $(TARG)
```

Though we won't go into the details of make syntax, as you can see, this makefile can make your life somewhat easier. For example, it allows

you to easily add new source files into your build, simply by adding them to the SRCS variable at the top of the makefile. You can also easily change the name of the executable by changing the TARG line, and the compiler, flags, and library specifications are all easily modified.

One final word about make: figuring out a target's prerequisites is not always trivial, especially in large and complex programs. Not surprisingly, there is another tool that helps with this, called makedepend. Read about it on your own and see if you can incorporate it into a makefile.

F.7 Debugging

Finally, after you have created a good build environment, and a correctly compiled program, you may find that your program is buggy. One way to fix the problem(s) is to think really hard — this method is sometimes successful, but often not. The problem is a lack of *information*; you just don't know exactly what is going on within the program, and therefore cannot figure out why it is not behaving as expected. Fortunately, there is some help: gdb, the GNU debugger.

Let's take the following buggy code, saved in the file buggy.c, and compiled into the executable buggy.

```
#include <stdio.h>

struct Data {
    int x;
};

int
main(int argc, char *argv[])
{
    struct Data *p = NULL;
    printf("%d\n", p->x);
}
```

In this example, the main program dereferences the variable p when it is NULL, which will lead to a segmentation fault. Of course, this problem should be easy to fix by inspection, but in a more complex program, finding such a problem is not always easy.

To prepare yourself for a debugging session, recompile your program and make sure to pass the -g flag to each compile line. This includes extra debugging information in your executable that will be useful during your debugging session. Also, don't turn on optimization (-O); though this may work, it may also lead to confusion during debugging.

After re-compiling with -g, you are ready to use the debugger. Fire up gdb at the command prompt as follows:

```
prompt> gdb buggy
```

This puts you inside an interactive session with the debugger. Note that you can also use the debugger to examine "core" files that were pro-

duced during bad runs, or to attach to an already-running program; read the documentation to learn more about this.

Once inside, you may see something like this:

```
prompt> gdb buggy
GNU gdb ...
Copyright 2008 Free Software Foundation, Inc.
(gdb)
```

The first thing you might want to do is to go ahead and run the program. To do this, simply type `run` at gdb command prompt. In this case, this is what you might see:

```
(gdb) run
Starting program: buggy

Program received signal SIGSEGV, Segmentation fault.
0x8048433 in main (argc=1, argv=0xbffff844) at buggy.cc:19
19          printf("%d\n", p->x);
```

As you can see from the example, in this case, `gdb` immediately pinpoints where the problem occurred; a "segmentation fault" was generated at the line where we tried to dereference `p`. This just means that we accessed some memory that we weren't supposed to access. At this point, the astute programmer can examine the code, and say "aha! it must be that `p` does not point to anything valid, and thus should not be dereferenced!", and then go ahead and fix the problem.

However, if you didn't know what was going on, you might want to examine some variable. `gdb` allows you to do this interactively during the debug session.

```
(gdb) print p
1 = (Data *) 0x0
```

By using the print primitive, we can examine `p`, and see both that it is a pointer to a struct of type Data, and that it is currently set to NULL (or zero, or hex zero which is shown here as "0x0").

Finally, you can also set breakpoints within your program to have the debugger stop the program at a certain routine. After doing this, it is often useful to step through the execution (one line at a time), and see what is happening.

```
(gdb) break main
Breakpoint 1 at 0x8048426: file buggy.cc, line 17.
(gdb) run
Starting program: /homes/hacker/buggy

Breakpoint 1, main (argc=1, argv=0xbffff844) at buggy.cc:17
17          struct Data *p = NULL;
(gdb) next
19          printf("%d\n", p->x);
(gdb)

Program received signal SIGSEGV, Segmentation fault.
0x8048433 in main (argc=1, argv=0xbffff844) at buggy.cc:19
19          printf("%d\n", p->x);
```

In the example above, a breakpoint is set at the `main()` routine; thus, when we run the program, the debugger almost immediately stops execution at main. At that point in the example, a "next" command is issued, which executes the next source-level command. Both "next" and "step" are useful ways to advance through a program — read about them in the documentation for more details [2].

This discussion really does not do `gdb` justice; it is a rich and flexible debugging tool, with many more features than can be described in the limited space here. Read more about it on your own and become an expert in your copious spare time.

F.8 Documentation

To learn a lot more about all of these things, you have to do two things: the first is to use these tools, and the second is to read more about them on your own. One way to find out more about `gcc`, `gmake`, and `gdb` is to read their man pages; type `man gcc`, `man gmake`, or `man gdb` at your command prompt. You can also use `man -k` to search the man pages for keywords, though that doesn't always work as well as it might; googling is probably a better approach here.

One tricky thing about man pages: typing `man XXX` may not result in the thing you want, if there is more than one thing called `XXX`. For example, if you are looking for the `kill()` system call man page, and if you just type `man kill` at the prompt, you will get the wrong man page, because there is a command-line program called `kill`. Man pages are divided into **sections**, and by default, man will return the man page in the lowest section that it finds, which in this case is section 1. Note that you can tell which man page you got by looking at the top of the page: if you see `kill(2)`, you know you are in the right man page in Section 2, where system calls live. Type `man man` to learn more about what is stored in each of the different sections of the man pages. Also note that `man -a kill` can be used to cycle through all of the different man pages named "kill".

Man pages are useful for finding out a number of things. In particular, you will often want to look up what arguments to pass to a library call, or what header files need to be included to use a library call. All of this should be available in the man page. For example, if you look up the `open()` system call, you will see:

```
SYNOPSIS
     #include <sys/types.h>
     #include <sys/stat.h>
     #include <fcntl.h>

     int open(const char *path, int oflag, /* mode_t mode */...);
```

[2] In particular, you can use the interactive "help" command while debugging with `gdb`

That tells you to include the headers `sys/types.h`, `sys/stat.h`, and `fcntl.h` in order to use the open call. It also tells you about the parameters to pass to open, namely a string called `path`, and integer flag `oflag`, and an optional argument to specify the `mode` of the file. If there were any libraries you needed to link with to use the call, it would tell you that here too.

Man pages require some effort to use effectively. They are often divided into a number of standard sections. The main body will describe how you can pass different parameters in order to have the function behave differently.

One particularly useful section is called the `RETURN VALUES` part of the man page, and it tells you what the function will return under success or failure. From the `open()` man page again:

```
RETURN VALUES
     Upon successful completion, the open() function opens the
     file and return a non-negative integer representing the
     lowest numbered unused file descriptor. Otherwise, -1 is
     returned, errno is set to indicate the error, and no files
     are created or modified.
```

Thus, by checking what open returns, you can see if the open succeeded or not. If it didn't, open (and many standard library routines) will set a global variable called `errno` to a value to tell you about the error. See the `ERRORS` section of the man page for more details.

Another thing you might want to do is to look for the definition of a structure that is not specified in the man page itself. For example, the man page for `gettimeofday()` has the following synopsis:

```
SYNOPSIS
     #include <sys/time.h>
     int gettimeofday(struct timeval *restrict tp,
                      void *restrict tzp);
```

From this page, you can see that the time is put into a structure of type `timeval`, but the man page may not tell you what fields that struct has! (in this case, it does, but you may not always be so lucky) Thus, you may have to hunt for it. All include files are found under the directory `/usr/include`, and thus you can use a tool like `grep` to look for it. For example, you might type:

```
prompt> grep 'struct timeval' /usr/include/sys/*.h
```

This lets you look for the definition of the structure in all files that end with `.h` in `/usr/include/sys`. Unfortunately, this may not always work, as that include file may include others which are found elsewhere.

A better way to do this is to use a tool at your disposal, the compiler. Write a program that includes the header `time.h`, let's say called `main.c`. Then, instead of compiling it, use the compiler to invoke the preprocessor. The preprocessor processes all the directives in your file, such as `#define` commands and `#include` commands. To do this, type

`gcc -E main.c`. The result of this is a C file that has all of the needed structures and prototypes in it, including the definition of the timeval struct.

Probably an even better way to find these things out: google. You should always google things you don't know about — it's amazing how much you can learn simply by looking it up!

Info Pages

Also quite useful in the hunt for documentation are the **info pages**, which provide much more detailed documentation on many GNU tools. You can access the info pages by running the program `info`, or via `emacs`, the preferred editor of hackers, by executing `Meta-x info`. A program like `gcc` has hundreds of flags, and some of them are surprisingly useful to know about. `gmake` has many more features that will improve your build environment. Finally, `gdb` is quite a sophisticated debugger. Read the man and info pages, try out features that you hadn't tried before, and become a power user of your programming tools.

F.9 Suggested Readings

Other than the man and info pages, there are a number of useful books out there. Note that a lot of this information is available for free on-line; however, sometimes having something in book form seems to make it easier to learn. Also, always look for O'Reilly books on topics you are interested in; they are almost always of high quality.

- "The C Programming Language", by Brian Kernighan and Dennis Ritchie. This is *the* definitive C book to have.
- "Managing Projects with make", by Andrew Oram and Steve Talbott. A reasonable and short book on make.
- "Debugging with GDB: The GNU Source-Level Debugger", by Richard M. Stallman, Roland H. Pesch. A little book on using GDB.
- "Advanced Programming in the UNIX Environment", by W. Richard Stevens and Steve Rago. Stevens wrote some excellent books, and this is a must for UNIX hackers. He also has an excellent set of books on TCP/IP and Sockets programming.
- "Expert C Programming", by Peter Van der Linden. A lot of the useful tips about compilers, etc., above are stolen directly from here. Read this! It is a great and eye-opening book, even though a little out of date.

G

Laboratory: Systems Projects

This chapter presents some ideas for systems projects. We usually do about six or seven projects in a 15-week semester, meaning one every two weeks or so. The first few are usually done by a single student, and the last few in groups of size two.

Each semester, the projects follow this same outline; however, we vary the details to keep it interesting and make "sharing" of code across semesters more challenging (not that anyone would do that!). We also use the Moss tool [M94] to look for this kind of "sharing".

As for grading, we've tried a number of different approaches, each of which have their strengths and weaknesses. Demos are fun but time consuming. Automated test scripts are less time intensive but require a great deal of care to get them to carefully test interesting corner cases. Check the book web page for more details on these projects; if you'd like the automated test scripts, we'd be happy to share.

G.1 Intro Project

The first project is an introduction to systems programming. Typical assignments have been to write some variant of the `sort` utility, with different constraints. For example, sorting text data, sorting binary data, and other similar projects all make sense. To complete the project, one must get familiar with some system calls (and their return error codes), use a few simple data structures, and not much else.

G.2 UNIX Shell

In this project, students build a variant of a UNIX shell. Students learn about process management as well as how mysterious things like pipes and redirects actually work. Variants include unusual features, like a redirection symbol that also compresses the output via gzip. Another variant is a batch mode which allows the user to batch up a few requests and then execute them, perhaps using different scheduling disciplines.

G.3 Memory-allocation Library

This project explores how a chunk of memory is managed, by building an alternative memory-allocation library (like `malloc()` and `free()` but with different names). The project teaches students how to use `mmap()` to get a chunk of anonymous memory, and then about pointers in great detail in order to build a simple (or perhaps, more complex) free list to manage the space. Variants include: best/worst fit, buddy, and various other allocators.

G.4 Intro to Concurrency

This project introduces concurrent programming with POSIX threads. Build some simple thread-safe libraries: a list, hash table, and some more complicated data structures are good exercises in adding locks to real-world code. Measure the performance of coarse-grained versus fine-grained alternatives. Variants just focus on different (and perhaps more complex) data structures.

G.5 Concurrent Web Server

This project explores the use of concurrency in a real-world application. Students take a simple web server (or build one) and add a thread pool to it, in order to serve requests concurrently. The thread pool should be of a fixed size, and use a producer/consumer bounded buffer to pass requests from a main thread to the fixed pool of workers. Learn how threads, locks, and condition variables are used to build a real server. Variants include scheduling policies for the threads.

G.6 File System Checker

This project explores on-disk data structures and their consistency. Students build a simple file system checker. The `debugfs` tool can be used on Linux to make real file-system images; crawl through them and make sure all is well. To make it more difficult, also fix any problems that are found. Variants focus on different types of problems: pointers, link counts, use of indirect blocks, etc.

G.7 File System Defragmenter

This project explores on-disk data structures and their performance implications. The project should give some particular file-system images to students with known fragmentation problems; students should then crawl through the image, and look for files that are not laid out sequentially. Write out a new "defragmented" image that fixes this problem, perhaps reporting some statistics.

G.8 Concurrent File Server

This project combines concurrency and file systems and even a little bit of networking and distributed systems. Students build a simple concurrent file server. The protocol should look something like NFS, with lookups, reads, writes, and stats. Store files within a single disk image (designed as a file). Variants are manifold, with different suggested on-disk formats and network protocols.

References

[M94] "Moss: A System for Detecting Software Plagiarism"
Alex Aiken
Available: http://theory.stanford.edu/~aiken/moss/

H

Laboratory: xv6 Projects

This chapter presents some ideas for projects related to the xv6 kernel. The kernel is available from MIT and is quite fun to play with; doing these projects also make the in-class material more directly relevant to the projects. These projects (except perhaps the first couple) are usually done in pairs, making the hard task of staring at the kernel a little easier.

H.1 Intro Project

The introduction adds a simple system call to xv6. Many variants are possible, including a system call to count how many system calls have taken place (one counter per system call), or other information-gathering calls. Students learn about how a system call actually takes place.

H.2 Processes and Scheduling

Students build a more complicated scheduler than the default round robin. Many variants are possible, including a Lottery scheduler or multi-level feedback queue. Students learn how schedulers actually work, as well as how a context switch takes place. A small addendum is to also require students to figure out how to make processes return a proper error code when exiting, and to be able to access that error code through the `wait()` system call.

H.3 Intro to Virtual Memory

The basic idea is to add a new system call that, given a virtual address, returns the translated physical address (or reports that the address is not valid). This lets students see how the virtual memory system sets up page tables without doing too much hard work. Another variant explores how to transform xv6 so that a null-pointer dereference actually generates a fault.

H.4 Copy-on-write Mappings

This project adds the ability to perform a lightweight `fork()`, called `vfork()`, to xv6. This new call doesn't simply copy the mappings but rather sets up copy-on-write mappings to shared pages. Upon reference to such a page, the kernel must then create a real copy and update page tables accordingly.

H.5 Memory mappings

An alternate virtual memory project is to add some form of memory-mapped files. Probably the easiest thing to do is to perform a lazy page-in of code pages from an executable; a more full-blown approach is to build an `mmap()` system call and all of the requisite infrastructure needed to fault in pages from disk upon dereference.

H.6 Kernel Threads

This project explores how to add kernel threads to xv6. A `clone()` system call operates much like fork but uses the same address space. Students have to figure out how to implement such a call, and thus how to create a real kernel thread. Students also should build a little thread library on top of that, providing simple locks.

H.7 Advanced Kernel Threads

Students build a full-blown thread library on top of their kernel threads, adding different types of locks (spin locks, locks that sleep when the processor is not available) as well as condition variables. Requisite kernel support is added as well.

H.8 Extent-based File System

This first file system project adds some simple features to the basic file system. For files of type EXTENT, students change the inode to store extents (i.e., pointer, length pairs) instead of just pointers. Serves as a relatively light introduction to the file system.

H.9 Fast File System

Students transform the basic xv6 file system into the Berkeley Fast File System (FFS). Students build a new `mkfs` tool, introduce block groups and a new block-allocation policy, and build the large-file exception. The basics of how file systems work are understood at a deeper level.

H.10 Journaling File System

Students add a rudimentary journaling layer to xv6. For each write to a file, the journaling FS batches up all dirtied blocks and writes a record of their pending update to an on-disk log; only then are the blocks modified in place. Students demonstrate the correctness of their system by introducing crash points and showing that the file system always recovers to a consistent state.

H.11 File System Checker

Students build a simple file system checker for the xv6 file system. Students learn about what makes a file system consistent and how exactly to check for it.

I

Flash-based SSDs

After decades of hard-disk drive dominance, a new form of persistent storage device has recently gained significance in the world. Generically referred to as **solid-state storage**, such devices have no mechanical or moving parts like hard drives; rather, they are simply built out of transistors, much like memory and processors. However, unlike typical random-access memory (e.g., DRAM), such a **solid-state storage device** (a.k.a., an **SSD**) retains information despite power loss, and thus is an ideal candidate for use in persistent storage of data.

The technology we'll focus on is known as **flash** (more specifically, **NAND-based flash**), which was created by Fujio Masuoka in the 1980s [M+14]. Flash, as we'll see, has some unique properties. For example, to write to a given chunk of it (i.e., a **flash page**), you first have to erase a bigger chunk (i.e., a **flash block**), which can be quite expensive. In addition, writing too often to a page will cause it to **wear out**. These two properties make construction of a flash-based SSD an interesting challenge:

CRUX: HOW TO BUILD A FLASH-BASED SSD

How can we build a flash-based SSD? How can we handle the expensive nature of erasing? How can we build a device that lasts a long time, given that repeated overwrite will wear the device out? Will the march of progress in technology ever cease? Or cease to amaze?

I.1 Storing a Single Bit

Flash chips are designed to store one or more bits in a single transistor; the level of charge trapped within the transistor is mapped to a binary value. In a **single-level cell** (**SLC**) flash, only a single bit is stored within a transistor (i.e., 1 or 0); with a **multi-level cell** (**MLC**) flash, two bits are encoded into different levels of charge, e.g., 00, 01, 10, and 11 are represented by low, somewhat low, somewhat high, and high levels. There is even **triple-level cell** (**TLC**) flash, which encodes 3 bits per cell. Overall, SLC chips achieve higher performance and are more expensive.

TIP: BE CAREFUL WITH TERMINOLOGY
You may have noticed that some terms we have used many times before (blocks, pages) are being used within the context of a flash, but in slightly different ways than before. New terms are not created to make your life harder (although they may be doing just that), but arise because there is no central authority where terminology decisions are made. What is a block to you may be a page to someone else, and vice versa, depending on the context. Your job is simple: to know the appropriate terms within each domain, and use them such that people well-versed in the discipline can understand what you are talking about. It's one of those times where the only solution is simple but sometimes painful: use your memory.

Of course, there are many details as to exactly how such bit-level storage operates, down at the level of device physics. While beyond the scope of this book, you can read more about it on your own [J10].

I.2 From Bits to Banks/Planes

As they say in ancient Greece, storing a single bit (or a few) does not a storage system make. Hence, flash chips are organized into **banks** or **planes** which consist of a large number of cells.

A bank is accessed in two different sized units: **blocks** (sometimes called **erase blocks**), which are typically of size 128 KB or 256 KB, and **pages**, which are a few KB in size (e.g., 4KB). Within each bank there are a large number of blocks; within each block, there are a large number of pages. When thinking about flash, you must remember this new terminology, which is different than the blocks we refer to in disks and RAIDs and the pages we refer to in virtual memory.

Figure I.1 shows an example of a flash plane with blocks and pages; there are three blocks, each containing four pages, in this simple example. We'll see below why we distinguish between blocks and pages; it turns out this distinction is critical for flash operations such as reading and writing, and even more so for the overall performance of the device. The most important (and weird) thing you will learn is that to write to a page within a block, you first have to erase the entire block; this tricky detail makes building a flash-based SSD an interesting and worthwhile challenge, and the subject of the second-half of the chapter.

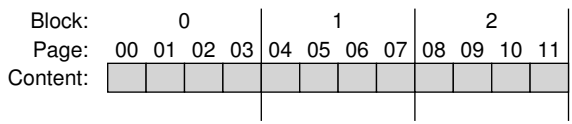


Figure I.1: **A Simple Flash Chip: Pages Within Blocks**

I.3 Basic Flash Operations

Given this flash organization, there are three low-level operations that a flash chip supports. The **read** command is used to read a page from the flash; **erase** and **program** are used in tandem to write. The details:

- **Read (a page)**: A client of the flash chip can read any page (e.g., 2KB or 4KB), simply by specifying the read command and appropriate page number to the device. This operation is typically quite fast, 10s of microseconds or so, regardless of location on the device, and (more or less) regardless of the location of the previous request (quite unlike a disk). Being able to access any location uniformly quickly means the device is a **random access** device.
- **Erase (a block):** Before writing to a *page* within a flash, the nature of the device requires that you first **erase** the entire *block* the page lies within. Erase, importantly, destroys the contents of the block (by setting each bit to the value 1); therefore, you must be sure that any data you care about in the block has been copied elsewhere (to memory, or perhaps to another flash block) *before* executing the erase. The erase command is quite expensive, taking a few milliseconds to complete. Once finished, the entire block is reset and each page is ready to be programmed.
- **Program (a page):** Once a block has been erased, the program command can be used to change some of the 1's within a page to 0's, and write the desired contents of a page to the flash. Programming a page is less expensive than erasing a block, but more costly than reading a page, usually taking around 100s of microseconds on modern flash chips.

One way to think about flash chips is that each page has a state associated with it. Pages start in an `INVALID` state. By erasing the block that a page resides within, you set the state of the page (and all pages within that block) to `ERASED`, which resets the content of each page in the block but also (importantly) makes them programmable. When you program a page, its state changes to `VALID`, meaning its contents have been set and can be read. Reads do not affect these states (although you should only read from pages that have been programmed). Once a page has been programmed, the only way to change its contents is to erase the entire block within which the page resides. Here is an example of states transition after various erase and program operations within a 4-page block:

		`iiii`	*Initial: pages in block are invalid* (`i`)
Erase()	→	`EEEE`	*State of pages in block set to erased* (`E`)
Program(0)	→	`VEEE`	*Program page 0; state set to valid* (`V`)
Program(0)	→	**error**	*Cannot re-program page after programming*
Program(1)	→	`VVEE`	*Program page 1*
Erase()	→	`EEEE`	*Contents erased; all pages programmable*

A Detailed Example

Because the process of writing (i.e., erasing and programming) is so unusual, let's go through a detailed example to make sure it makes sense. In this example, imagine we have the following four 8-bit pages, within a 4-page block (both unrealistically small sizes, but useful within this example); each page is `VALID` as each has been previously programmed.

Page 0	Page 1	Page 2	Page 3
00011000	11001110	00000001	00111111
VALID	VALID	VALID	VALID

Now say we wish to write to page 0, filling it with new contents. To write any page, we must first erase the entire block. Let's assume we do so, thus leaving the block in this state:

Page 0	Page 1	Page 2	Page 3
11111111	11111111	11111111	11111111
ERASED	ERASED	ERASED	ERASED

Good news! We could now go ahead and program page 0, for example with the contents `00000011`, overwriting the old page 0 (contents `00011000`) as desired. After doing so, our block looks like this:

Page 0	Page 1	Page 2	Page 3
00000011	11111111	11111111	11111111
VALID	ERASED	ERASED	ERASED

And now the bad news: the previous contents of pages 1, 2, and 3 are all gone! Thus, before overwriting any page *within* a block, we must first move any data we care about to another location (e.g., memory, or elsewhere on the flash). The nature of erase will have a strong impact on how we design flash-based SSDs, as we'll soon learn about.

Summary

To summarize, reading a page is easy: just read the page. Flash chips do this quite well, and quickly; in terms of performance, they offer the potential to greatly exceed the random read performance of modern disk drives, which are slow due to mechanical seek and rotation costs.

Writing a page is trickier; the entire block must first be erased (taking care to first move any data we care about to another location), and then the desired page programmed. Not only is this expensive, but frequent repetitions of this program/erase cycle can lead to the biggest reliability problem flash chips have: **wear out**. When designing a storage system with flash, the performance and reliability of writing is a central focus. We'll soon learn more about how modern SSDs attack these issues, delivering excellent performance and reliability despite these limitations.

Device	Read (μs)	Program (μs)	Erase (μs)
SLC	25	200-300	1500-2000
MLC	50	600-900	~3000
TLC	~75	~900-1350	~4500

Figure I.2: **Raw Flash Performance Characteristics**

I.4 Flash Performance And Reliability

Because we're interested in building a storage device out of raw flash chips, it is worthwhile to understand their basic performance characteristics. Figure I.2 presents a rough summary of some numbers found in the popular press [V12]. Therein, the author presents the basic operation latency of reads, programs, and erases across SLC, MLC, and TLC flash, which store 1, 2, and 3 bits of information per cell, respectively.

As we can see from the table, read latencies are quite good, taking just 10s of microseconds to complete. Program latency is higher and more variable, as low as 200 microseconds for SLC, but higher as you pack more bits into each cell; to get good write performance, you will have to make use of multiple flash chips in parallel. Finally, erases are quite expensive, taking a few milliseconds typically. Dealing with this cost is central to modern flash storage design.

Let's now consider reliability of flash chips. Unlike mechanical disks, which can fail for a wide variety of reasons (including the gruesome and quite physical **head crash**, where the drive head actually makes contact with the recording surface), flash chips are pure silicon and in that sense have fewer reliability issues to worry about. The primary concern is **wear out**; when a flash block is erased and programmed, it slowly accrues a little bit of extra charge. Over time, as that extra charge builds up, it becomes increasingly difficult to differentiate between a 0 and a 1. At the point where it becomes impossible, the block becomes unusable.

The typical lifetime of a block is currently not well known. Manufacturers rate MLC-based blocks as having a 10,000 P/E (Program/Erase) cycle lifetime; that is, each block can be erased and programmed 10,000 times before failing. SLC-based chips, because they store only a single bit per transistor, are rated with a longer lifetime, usually 100,000 P/E cycles. However, recent research has shown that lifetimes are much longer than expected [BD10].

One other reliability problem within flash chips is known as **disturbance**. When accessing a particular page within a flash, it is possible that some bits get flipped in neighboring pages; such bit flips are known as **read disturbs** or **program disturbs**, depending on whether the page is being read or programmed, respectively.

TIP: THE IMPORTANCE OF BACKWARDS COMPATIBILITY
Backwards compatibility is always a concern in layered systems. By defining a stable interface between two systems, one enables innovation on each side of the interface while ensuring continued interoperability. Such an approach has been quite successful in many domains: operating systems have relatively stable APIs for applications, disks provide the same block-based interface to file systems, and each layer in the IP networking stack provides a fixed unchanging interface to the layer above.

Not surprisingly, there can be a downside to such rigidity, as interfaces defined in one generation may not be appropriate in the next. In some cases, it may be useful to think about redesigning the entire system entirely. An excellent example is found in the Sun ZFS file system [B07]; by reconsidering the interaction of file systems and RAID, the creators of ZFS envisioned (and then realized) a more effective integrated whole.

I.5 From Raw Flash to Flash-Based SSDs

Given our basic understanding of flash chips, we now face our next task: how to turn a basic set of flash chips into something that looks like a typical storage device. The standard storage interface is a simple block-based one, where blocks (sectors) of size 512 bytes (or larger) can be read or written, given a block address. The task of the flash-based SSD is to provide that standard block interface atop the raw flash chips inside it.

Internally, an SSD consists of some number of flash chips (for persistent storage). An SSD also contains some amount of volatile (i.e., non-persistent) memory (e.g., SRAM); such memory is useful for caching and buffering of data as well as for mapping tables, which we'll learn about below. Finally, an SSD contains control logic to orchestrate device operation. See Agrawal et. al for details [A+08]; a simplified block diagram is seen in Figure I.3 (page 7).

One of the essential functions of this control logic is to satisfy client reads and writes, turning them into internal flash operations as need be. The **flash translation layer**, or **FTL**, provides exactly this functionality. The FTL takes read and write requests on *logical blocks* (that comprise the device interface) and turns them into low-level read, erase, and program commands on the underlying *physical blocks* and *physical pages* (that comprise the actual flash device). The FTL should accomplish this task with the goal of delivering excellent performance and high reliability.

Excellent performance, as we'll see, can be realized through a combination of techniques. One key will be to utilize multiple flash chips in **parallel**; although we won't discuss this technique much further, suffice it to say that all modern SSDs use multiple chips internally to obtain higher performance. Another performance goal will be to reduce **write amplification**, which is defined as the total write traffic (in bytes) issued to the flash chips by the FTL divided by the total write traffic (in bytes) is-

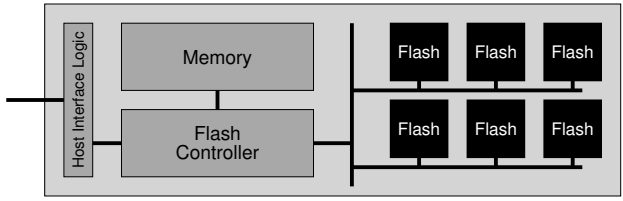


Figure I.3: **A Flash-based SSD: Logical Diagram**

sued by the client to the SSD. As we'll see below, naive approaches to FTL construction will lead to high write amplification and low performance.

High reliability will be achieved through the combination of a few different approaches. One main concern, as discussed above, is **wear out**. If a single block is erased and programmed too often, it will become unusable; as a result, the FTL should try to spread writes across the blocks of the flash as evenly as possible, ensuring that all of the blocks of the device wear out at roughly the same time; doing so is called **wear leveling** and is an essential part of any modern FTL.

Another reliability concern is program disturbance. To minimize such disturbance, FTLs will commonly program pages within an erased block *in order*, from low page to high page. This sequential-programming approach minimizes disturbance and is widely utilized.

I.6 FTL Organization: A Bad Approach

The simplest organization of an FTL would be something we call **direct mapped**. In this approach, a read to logical page N is mapped directly to a read of physical page N. A write to logical page N is more complicated; the FTL first has to read in the entire block that page N is contained within; it then has to erase the block; finally, the FTL programs the old pages as well as the new one.

As you can probably guess, the direct-mapped FTL has many problems, both in terms of performance as well as reliability. The performance problems come on each write: the device has to read in the entire block (costly), erase it (quite costly), and then program it (costly). The end result is severe write amplification (proportional to the number of pages in a block) and as a result, terrible write performance, even slower than typical hard drives with their mechanical seeks and rotational delays.

Even worse is the reliability of this approach. If file system metadata or user file data is repeatedly overwritten, the same block is erased and programmed, over and over, quickly wearing it out and potentially losing data. The direct mapped approach simply gives too much control over wear out to the client workload; if the workload does not spread write load evenly across its logical blocks, the underlying physical blocks containing popular data will quickly wear out. For both reliability and performance reasons, a direct-mapped FTL is a bad idea.

I.7 A Log-Structured FTL

For these reasons, most FTLs today are **log structured**, an idea useful in both storage devices (as we'll see now) and file systems above them (as we'll see in the chapter on **log-structured file systems**). Upon a write to logical block N, the device appends the write to the next free spot in the currently-being-written-to block; we call this style of writing **logging**. To allow for subsequent reads of block N, the device keeps a **mapping table** (in its memory, and persistent, in some form, on the device); this table stores the physical address of each logical block in the system.

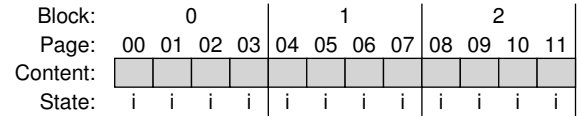
Let's go through an example to make sure we understand how the basic log-based approach works. To the client, the device looks like a typical disk, in which it can read and write 512-byte sectors (or groups of sectors). For simplicity, assume that the client is reading or writing 4-KB sized chunks. Let us further assume that the SSD contains some large number of 16-KB sized blocks, each divided into four 4-KB pages; these parameters are unrealistic (flash blocks usually consist of more pages) but will serve our didactic purposes quite well.

Assume the client issues the following sequence of operations:

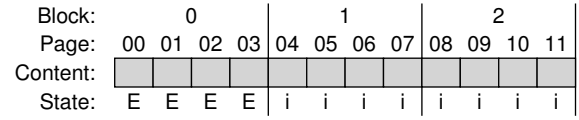
- Write(100) with contents `a1`
- Write(101) with contents `a2`
- Write(2000) with contents `b1`
- Write(2001) with contents `b2`

These **logical block addresses** (e.g., 100) are used by the client of the SSD (e.g., a file system) to remember where information is located.

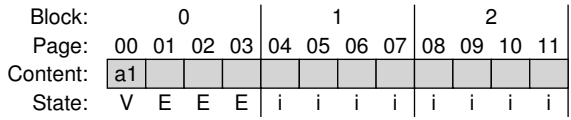
Internally, the device must transform these block writes into the erase and program operations supported by the raw hardware, and somehow record, for each logical block address, which **physical page** of the SSD stores its data. Assume that all blocks of the SSD are currently not valid, and must be erased before any page can be programmed. Here we show the initial state of our SSD, with all pages marked `INVALID` (`i`):



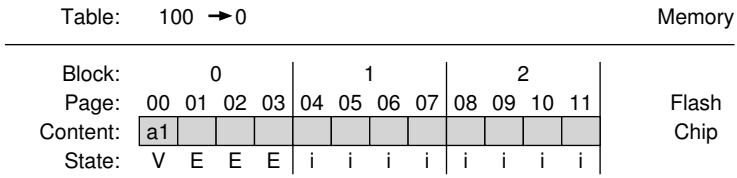
When the first write is received by the SSD (to logical block 100), the FTL decides to write it to physical block 0, which contains four physical pages: 0, 1, 2, and 3. Because the block is not erased, we cannot write to it yet; the device must first issue an erase command to block 0. Doing so leads to the following state:



Block 0 is now ready to be programmed. Most SSDs will write pages in order (i.e., low to high), reducing reliability problems related to **program disturbance**. The SSD then directs the write of logical block 100 into physical page 0:

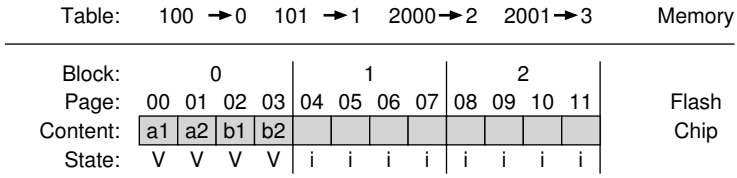


But what if the client wants to *read* logical block 100? How can it find where it is? The SSD must transform a read issued to logical block 100 into a read of physical page 0. To accommodate such functionality, when the FTL writes logical block 100 to physical page 0, it records this fact in an **in-memory mapping table**. We will track the state of this mapping table in the diagrams as well:



Now you can see what happens when the client writes to the SSD. The SSD finds a location for the write, usually just picking the next free page; it then programs that page with the block's contents, and records the logical-to-physical mapping in its mapping table. Subsequent reads simply use the table to **translate** the logical block address presented by the client into the physical page number required to read the data.

Let's now examine the rest of the writes in our example write stream: 101, 2000, and 2001. After writing these blocks, the state of the device is:



The log-based approach by its nature improves performance (erases only being required once in a while, and the costly read-modify-write of the direct-mapped approach avoided altogether), and greatly enhances reliability. The FTL can now spread writes across all pages, performing what is called **wear leveling** and increasing the lifetime of the device; we'll discuss wear leveling further below.

ASIDE: **FTL MAPPING INFORMATION PERSISTENCE**

You might be wondering: what happens if the device loses power? Does the in-memory mapping table disappear? Clearly, such information cannot truly be lost, because otherwise the device would not function as a persistent storage device. An SSD must have some means of recovering mapping information.

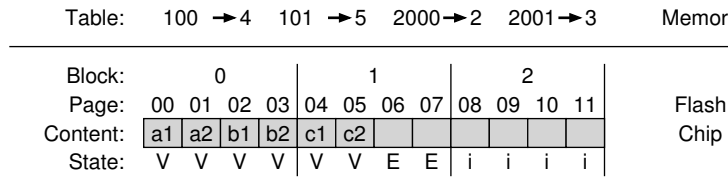
The simplest thing to do is to record some mapping information with each page, in what is called an **out-of-band** (**OOB**) area. When the device loses power and is restarted, it must reconstruct its mapping table by scanning the OOB areas and reconstructing the mapping table in memory. This basic approach has its problems; scanning a large SSD to find all necessary mapping information is slow. To overcome this limitation, some higher-end devices use more complex **logging** and **checkpointing** techniques to speed up recovery; we'll learn more about logging later when we discuss file systems.

Unfortunately, this basic approach to log structuring has some downsides. The first is that overwrites of logical blocks lead to something we call **garbage**, i.e., old versions of data around the drive and taking up space. The device has to periodically perform **garbage collection** (**GC**) to find said blocks and free space for future writes; excessive garbage collection drives up write amplification and lowers performance. The second is high cost of in-memory mapping tables; the larger the device, the more memory such tables need. We now discuss each in turn.

I.8 Garbage Collection

The first cost of any log-structured approach such as this one is that garbage is created, and therefore **garbage collection** (i.e., dead-block reclamation) must be performed. Let's use our continued example to make sense of this. Recall that logical blocks 100, 101, 2000, and 2001 have been written to the device.

Now, let's assume that blocks 100 and 101 are written to again, with contents `c1` and `c2`. The writes are written to the next free pages (in this case, physical pages 4 and 5), and the mapping table is updated accordingly. Note that the device must have first erased block 1 to make such programming possible:



The problem we have now should be obvious: physical pages 0 and 1, although marked `VALID`, have **garbage** in them, i.e., the old versions of blocks 100 and 101. Because of the log-structured nature of the device, overwrites create garbage blocks, which the device must reclaim to provide free space for new writes to take place.

The process of finding garbage blocks (also called **dead blocks**) and reclaiming them for future use is called **garbage collection**, and it is an important component of any modern SSD. The basic process is simple: find a block that contains one or more garbage pages, read in the live (non-garbage) pages from that block, write out those live pages to the log, and (finally) reclaim the entire block for use in writing.

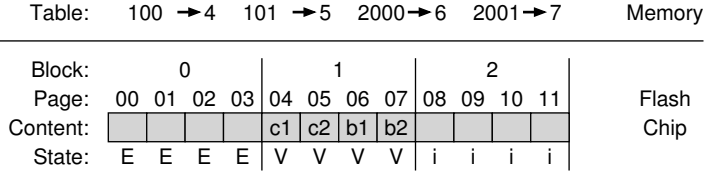
Let's now illustrate with an example. The device decides it wants to reclaim any dead pages within block 0 above. Block 0 has two dead blocks (pages 0 and 1) and two lives blocks (pages 2 and 3, which contain blocks 2000 and 2001, respectively). To do so, the device will:

- Read live data (pages 2 and 3) from block 0
- Write live data to end of the log
- Erase block 0 (freeing it for later usage)

For the garbage collector to function, there must be enough information within each block to enable the SSD to determine whether each page is live or dead. One natural way to achieve this end is to store, at some location within each block, information about which logical blocks are stored within each page. The device can then use the mapping table to determine whether each page within the block holds live data or not.

From our example above (before the garbage collection has taken place), block 0 held logical blocks 100, 101, 2000, 2001. By checking the mapping table (which, before garbage collection, contained `100->4, 101->5, 2000->2, 2001->3`), the device can readily determine whether each of the pages within the SSD block holds live information. For example, 2000 and 2001 clearly are still pointed to by the map; 100 and 101 are not and therefore are candidates for garbage collection.

When this garbage collection process is complete in our example, the state of the device is:



As you can see, garbage collection can be expensive, requiring reading and rewriting of live data. The ideal candidate for reclamation is a block that consists of only dead pages; in this case, the block can immediately be erased and used for new data, without expensive data migration.

To reduce GC costs, some SSDs **overprovision** the device [A+08]; by adding extra flash capacity, cleaning can be delayed and pushed to the **background**, perhaps done at a time when the device is less busy. Adding more capacity also increases internal bandwidth, which can be used for cleaning and thus not harm perceived bandwidth to the client. Many modern drives overprovision in this manner, one key to achieving excellent overall performance.

I.9 Mapping Table Size

The second cost of log-structuring is the potential for extremely large mapping tables, with one entry for each 4-KB page of the device. With a large 1-TB SSD, for example, a single 4-byte entry per 4-KB page results in 1 GB of memory needed the device, just for these mappings! Thus, this **page-level** FTL scheme is impractical.

Block-Based Mapping

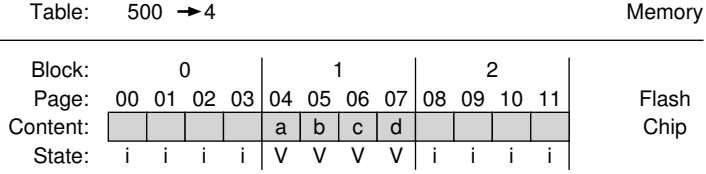
One approach to reduce the costs of mapping is to only keep a pointer per *block* of the device, instead of per page, reducing the amount of mapping information by a factor of $\frac{Size_{block}}{Size_{page}}$. This **block-level** FTL is akin to having bigger page sizes in a virtual memory system; in that case, you use fewer bits for the VPN and have a larger offset in each virtual address.

Unfortunately, using a block-based mapping inside a log-based FTL does not work very well for performance reasons. The biggest problem arises when a "small write" occurs (i.e., one that is less than the size of a physical block). In this case, the FTL must read a large amount of live data from the old block and copy it into a new one (along with the data from the small write). This data copying increases write amplification greatly and thus decreases performance.

To make this issue more clear, let's look at an example. Assume the client previously wrote out logical blocks 2000, 2001, 2002, and 2003 (with contents, `a`, `b`, `c`, `d`), and that they are located within physical block 1 at physical pages 4, 5, 6, and 7. With per-page mappings, the translation table would have to record four mappings for these logical blocks: 2000→4, 2001→5, 2002→6, 2003→7.

If, instead, we use block-level mapping, the FTL only need only to record a single address translation for all of this data. The address mapping, however, is slightly different than our previous examples. Specifically, we think of the logical address space of the device as being chopped into chunks that are the size of the physical blocks within the flash. Thus, the logical block address consists of two portions: a chunk number and an offset. Because we are assuming four logical blocks fit within each physical block, the offset portion of the logical addresses requires 2 bits; the remaining (most significant) bits form the chunk number.

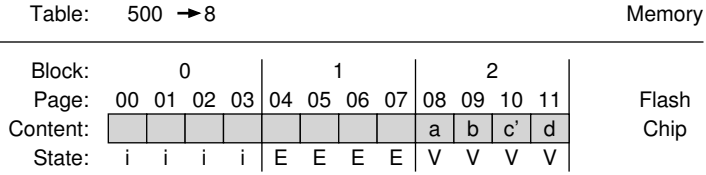
Logical blocks 2000, 2001, 2002, and 2003 all have the same chunk number (500), and have different offsets (0, 1, 2, and 3, respectively). Thus, with a block-level mapping, the FTL records that chunk 500 maps to block 1 (starting at physical page 4), as shown in this diagram:



In a block-based FTL, reading is easy. First, the FTL extracts the chunk number from the logical block address presented by the client, by taking the topmost bits out of the address. Then, the FTL looks up the chunk-number to physical-page mapping in the table. Finally, the FTL computes the address of the desired flash page by *adding* the offset from the logical address to the physical address of the block.

For example, if the client issues a read to logical address 2002, the device extracts the logical chunk number (500), looks up the translation in the mapping table (finding 4), and adds the offset from the logical address (2) to the translation (4). The resulting physical-page address (6) is where the data is located; the FTL can then issue the read to that physical address and obtain the desired data (c).

But what if the client writes to logical block 2002 (with contents c')? In this case, the FTL must read in 2000, 2001, and 2003, and then write out all four logical blocks in a new location, updating the mapping table accordingly. Block 1 (where the data used to reside) can then be erased and reused, as shown here.



As you can see from this example, while block level mappings greatly reduce the amount of memory needed for translations, they cause significant performance problems when writes are smaller than the physical block size of the device; as real physical blocks can be 256KB or larger, such writes are likely to happen quite often. Thus, a better solution is needed. Can you sense that this is the part of the chapter where we tell you what that solution is? Better yet, can you figure it out yourself, before reading on?

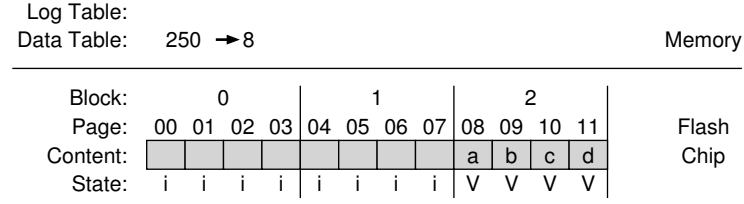
Hybrid Mapping

To enable flexible writing but also reduce mapping costs, many modern FTLs employ a **hybrid mapping** technique. With this approach, the FTL keeps a few blocks erased and directs all writes to them; these are called **log blocks**. Because the FTL wants to be able to write any page to any location within the log block without all the copying required by a pure block-based mapping, it keeps *per-page* mappings for these log blocks.

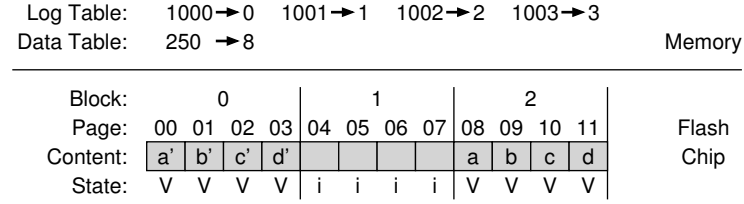
The FTL thus logically has two types of mapping table in its memory: a small set of per-page mappings in what we'll call the *log table*, and a larger set of per-block mappings in the *data table*. When looking for a particular logical block, the FTL will first consult the log table; if the logical block's location is not found there, the FTL will then consult the data table to find its location and then access the requested data.

The key to the hybrid mapping strategy is keeping the number of log blocks small. To keep the number of log blocks small, the FTL has to periodically examine log blocks (which have a pointer per page) and *switch* them into blocks that can be pointed to by only a single block pointer. This switch is accomplished by one of three main techniques, based on the contents of the block [KK+02].

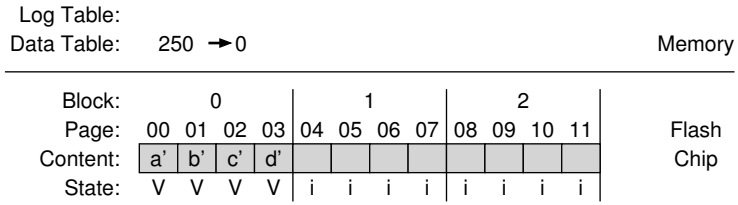
For example, let's say the FTL had previously written out logical pages 1000, 1001, 1002, and 1003, and placed them in physical block 2 (physical pages 8, 9, 10, 11); assume the contents of the writes to 1000, 1001, 1002, and 1003 are `a`, `b`, `c`, and `d`, respectively.



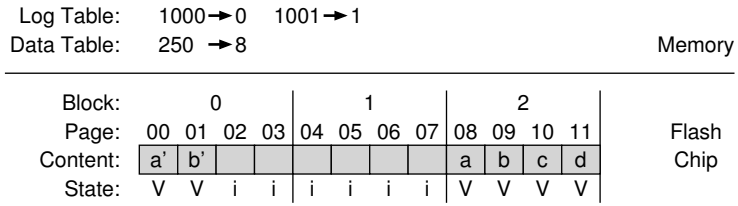
Now assume that the client overwrites each of these pages (with data `a'`, `b'`, `c'`, and `d'`), in the exact same order, in one of the currently available log blocks, say physical block 0 (physical pages 0, 1, 2, and 3). In this case, the FTL will have the following state:



Because these blocks have been written exactly in the same manner as before, the FTL can perform what is known as a **switch merge**. In this case, the log block (0) now becomes the storage location for pages 0, 1, 2, and 3, and is pointed to by a single block pointer; the old block (2) is now erased and used as a log block. In this best case, all the per-page pointers required replaced by a single block pointer.



This switch merge is the best case for a hybrid FTL. Unfortunately, sometimes the FTL is not so lucky. Imagine the case where we have the same initial conditions (logical blocks 0, 1, 2, and 4 stored in physical block 2) but then the client overwrites only logical blocks 0 and 1:



To reunite the other pages of this physical block, and thus be able to refer to them by only a single block pointer, the FTL performs what is called a **partial merge**. In this operation, 2 and 3 are read from block 4, and then appended to the log. The resulting state of the SSD is the same as the switch merge above; however, in this case, the FTL had to perform extra I/O to achieve its goals (in this case, reading logical blocks 2 and 3 from physical pages 18 and 19, and then writing them out to physical pages 22 and 23), thus increasing write amplification.

The final case encountered by the FTL known as a **full merge**, and requires even more work. In this case, the FTL must pull together pages from many other blocks to perform cleaning. For example, imagine that pages 0, 4, 8, and 12 are written to log block A. To switch this log block into a block-mapped page, the FTL must first create a data block containing logical blocks 0, 1, 2, and 3, and thus the FTL must read 1, 2, and 3 from elsewhere and then write out 0, 1, 2, and 3 together. Next, the merge must do the same for logical block 4, finding 5, 6, and 7 and reconciling them into a single data block. The same must be done for logical blocks 8 and 12, and then (finally), the log block A can be freed. Frequent full merges, as is not surprising, can seriously hurt performance [GY+09].

I.10 Wear Leveling

Finally, a related background activity that modern FTLs must implement is **wear leveling**, as introduced above. The basic idea is simple: because multiple erase/program cycles will wear out a flash block, the FTL should try its best to spread that work across all the blocks of the device evenly. In this manner, all blocks will wear out at roughly the same time, instead of a few "popular" blocks quickly unusable.

The basic log-structuring approach does a good initial job of spreading out write load, and garbage collection helps as well. However, sometimes a block will be filled with long-lived data that does not get over-written; in this case, garbage collection will never reclaim the block, and thus it does not receive its fair share of the write load.

To remedy this problem, the FTL must periodically read all the live data out of such blocks and re-write it elsewhere, thus making the block available for writing again. This process of wear leveling increases the write amplification of the SSD, and thus decreases performance as extra I/O is required to ensure that all blocks wear at roughly the same rate. Many different algorithms exist in the literature [A+08, M+14]; read more if you are interested.

I.11 SSD Performance And Cost

Before closing, let's examine the performance and cost of modern SSDs, to better understand how they will likely be used in persistent storage systems. In both cases, we'll compare to classic hard-disk drives (HDDs), and highlight the biggest differences between the two.

Performance

Unlike hard disk drives, flash-based SSDs have no mechanical components, and in fact are in many ways more similar to DRAM, in that they are "random access" devices. The biggest difference in performance, as compared to disk drives, is realized when performing random reads and writes; while a typical disk drive can only perform a few hundred random I/Os per second, SSDs can do much better. Here, we use some data from modern SSDs to see just how much better SSDs perform; we're particularly interested in how well the FTLs hide the performance issues of the raw chips.

Table I.4 shows some performance data for three different SSDs and one top-of-the-line hard drive; the data was taken from a few different online sources [S13, T15]. The left two columns show random I/O performance, and the right two columns sequential; the first three rows show data for three different SSDs (from Samsung, Seagate, and Intel), and the last row shows performance for a **hard disk drive** (or **HDD**), in this case a Seagate high-end drive.

Device	Random Reads (MB/s)	Random Writes (MB/s)	Sequential Reads (MB/s)	Sequential Writes (MB/s)
Samsung 840 Pro SSD	103	287	421	384
Seagate 600 SSD	84	252	424	374
Intel SSD 335 SSD	39	222	344	354
Seagate Savvio 15K.3 HDD	2	2	223	223

Figure I.4: **SSDs And Hard Drives: Performance Comparison**

We can learn a few interesting facts from the table. First, and most dramatic, is the difference in random I/O performance between the SSDs and the lone hard drive. While the SSDs obtain tens or even hundreds of MB/s in random I/Os, this "high performance" hard drive has a peak of just a couple MB/s (in fact, we rounded up to get to 2 MB/s). Second, you can see that in terms of sequential performance, there is much less of a difference; while the SSDs perform better, a hard drive is still a good choice if sequential performance is all you need. Third, you can see that SSD random read performance is not as good as SSD random write performance. The reason for such unexpectedly good random-write performance is due to the log-structured design of many SSDs, which transforms random writes into sequential ones and improves performance. Finally, because SSDs exhibit some performance difference between sequential and random I/Os, many of the techniques we will learn in subsequent chapters about how to build file systems for hard drives are still applicable to SSDs; although the magnitude of difference between sequential and random I/Os is smaller, there is enough of a gap to carefully consider how to design file systems to reduce random I/Os.

Cost

As we saw above, the performance of SSDs greatly outstrips modern hard drives, even when performing sequential I/O. So why haven't SSDs completely replaced hard drives as the storage medium of choice? The answer is simple: cost, or more specifically, cost per unit of capacity. Currently [A15], an SSD costs something like $150 for a 250-GB drive; such an SSD costs 60 cents per GB. A typical hard drive costs roughly $50 for 1-TB of storage, which means it costs 5 cents per GB. There is still more than a 10× difference in cost between these two storage media.

These performance and cost differences dictate how large-scale storage systems are built. If performance is the main concern, SSDs are a terrific choice, particularly if random read performance is important. If, on the other hand, you are assembling a large data center and wish to store massive amounts of information, the large cost difference will drive you towards hard drives. Of course, a hybrid approach can make sense – some storage systems are being assembled with both SSDs and hard drives, using a smaller number of SSDs for more popular "hot" data and delivering high performance, while storing the rest of the "colder" (less used) data on hard drives to save on cost. As long as the price gap exists, hard drives are here to stay.

I.12 Summary

Flash-based SSDs are becoming a common presence in laptops, desktops, and servers inside the datacenters that power the world's economy. Thus, you should probably know something about them, right?

Here's the bad news: this chapter (like many in this book) is just the first step in understanding the state of the art. Some places to get some more information about the raw technology include research on actual device performance (such as that by Chen et al. [CK+09] and Grupp et al. [GC+09]), issues in FTL design (including works by Agrawal et al. [A+08], Gupta et al. [GY+09], Huang et al. [H+14], Kim et al. [KK+02], Lee et al. [L+07], and Zhang et al. [Z+12]), and even distributed systems comprised of flash (including Gordon [CG+09] and CORFU [B+12]).

Don't just read academic papers; also read about recent advances in the popular press (e.g., [V12]). Therein you'll learn more practical (but still useful) information, such as Samsung's use of both TLC and SLC cells within the same SSD to maximize performance (SLC can buffer writes quickly) as well as capacity (TLC can store more bits per cell). And this is, as they say, just the tip of the iceberg. Dive in and learn more about this "iceberg" of research on your own, perhaps starting with Ma et al.'s excellent (and recent) survey [M+14]. Be careful though; icebergs can sink even the mightiest of ships [W15].

References

[A+08] "Design Tradeoffs for SSD Performance"
N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy
USENIX '08, San Diego California, June 2008
An excellent overview of what goes into SSD design.

[A15] "Amazon Pricing Study"
Remzi Arpaci-Dusseau
February, 2015
This is not an actual paper, but rather one of the authors going to Amazon and looking at current prices of hard drives and SSDs. You too can repeat this study, and see what the costs are today. Do it!

[B+12] "CORFU: A Shared Log Design for Flash Clusters"
M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, J. D. Davis
NSDI '12, San Jose, California, April 2012
A new way to think about designing a high-performace replicated log for clusters using Flash.

[BD10] "Write Endurance in Flash Drives: Measurements and Analysis"
Simona Boboila, Peter Desnoyers
FAST '10, San Jose, California, February 2010
A cool paper that reverse engineers flash-device lifetimes. Endurance sometimes far exceeds manufacturer predictions, by up to 100×.

[B07] "ZFS: The Last Word in File Systems"
Jeff Bonwick and Bill Moore
Available: http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf
Was this the last word in file systems? No, but maybe it's close.

[CG+09] "Gordon: Using Flash Memory to Build Fast,
Power-efficient Clusters for Data-intensive Applications"
Adrian M. Caulfield, Laura M. Grupp, Steven Swanson
ASPLOS '09, Washington, D.C., March 2009
Early research on assembling flash into larger-scale clusters; definitely worth a read.

[CK+09] "Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives"
Feng Chen, David A. Koufaty, and Xiaodong Zhang
SIGMETRICS/Performance '09, Seattle, Washington, June 2009
An excellent overview of SSD performance problems circa 2009 (though now a little dated).

[G14] "The SSD Endurance Experiment"
Geoff Gasior
The Tech Report, September 19, 2014
Available: http://techreport.com/review/27062
A nice set of simple experiments measuring performance of SSDs over time. There are many other similar studies; use google to find more.

[GC+09] "Characterizing Flash Memory: Anomalies, Observations, and Applications"
L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, J. K. Wolf
IEEE MICRO '09, New York, New York, December 2009
Another excellent characterization of flash performance.

[GY+09] "DFTL: a Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings"
Aayush Gupta, Youngjae Kim, Bhuvan Urgaonkar
ASPLOS '09, Washington, D.C., March 2009
This paper gives an excellent overview of different strategies for cleaning within hybrid SSDs as well as a new scheme which saves mapping table space and improves performance under many workloads.

[H+14] "An Aggressive Worn-out Flash Block Management Scheme
To Alleviate SSD Performance Degradation"
Ping Huang, Guanying Wu, Xubin He, Weijun Xiao
EuroSys '14, 2014
Recent work showing how to really get the most out of worn-out flash blocks; neat!

[J10] "Failure Mechanisms and Models for Semiconductor Devices"
Report JEP122F, November 2010
Available: http://www.jedec.org/sites/default/files/docs/JEP122F.pdf
A highly detailed discussion of what is going on at the device level and how such devices fail. Only for those not faint of heart. Or physicists. Or both.

[KK+02] "A Space-Efficient Flash Translation Layer For Compact Flash Systems"
Jesung Kim, Jong Min Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho
IEEE Transactions on Consumer Electronics, Volume 48, Number 2, May 2002
One of the earliest proposals to suggest hybrid mappings.

[L+07] "A Log Buffer-Based Flash Translation Layer
Using Fully-Associative Sector Translation"
Sang-won Lee, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, Ha-Joo Song
ACM Transactions on Embedded Computing Systems, Volume 6, Number 3, July 2007
A terrific paper about how to build hybrid log/block mappings.

[M+14] "A Survey of Address Translation Technologies for Flash Memories"
Dongzhe Ma, Jianhua Feng, Guoliang Li
ACM Computing Surveys, Volume 46, Number 3, January 2014
Probably the best recent survey of flash and related technologies.

[S13] "The Seagate 600 and 600 Pro SSD Review"
Anand Lal Shimpi
AnandTech, May 7, 2013
Available: http://www.anandtech.com/show/6935/seagate-600-ssd-review
One of many SSD performance measurements available on the internet. Haven't heard of the internet? No problem. Just go to your web browser and type "internet" into the search tool. You'll be amazed at what you can learn.

[T15] "Performance Charts Hard Drives"
Tom's Hardware, January 2015
Available: http://www.tomshardware.com/charts/enterprise-hdd-charts/
Yet another site with performance data, this time focusing on hard drives.

[V12] "Understanding TLC Flash"
Kristian Vatto
AnandTech, September, 2012
Available: http://www.anandtech.com/show/5067/understanding-tlc-nand
A short description about TLC flash and its characteristics.

[W15] "List of Ships Sunk by Icebergs"
Available: http://en.wikipedia.org/wiki/List_of_ships_sunk_by_icebergs
Yes, there is a wikipedia page about ships sunk by icebergs. It is a really boring page and basically everyone knows the only ship the iceberg-sinking-mafia cares about is the Titanic.

[Z+12] "De-indirection for Flash-based SSDs with Nameless Writes"
Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
FAST '13, San Jose, California, February 2013
Our research on a new idea to reduce mapping table space; the key is to re-use the pointers in the file system above to store locations of blocks, instead of adding another level of indirection.