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.

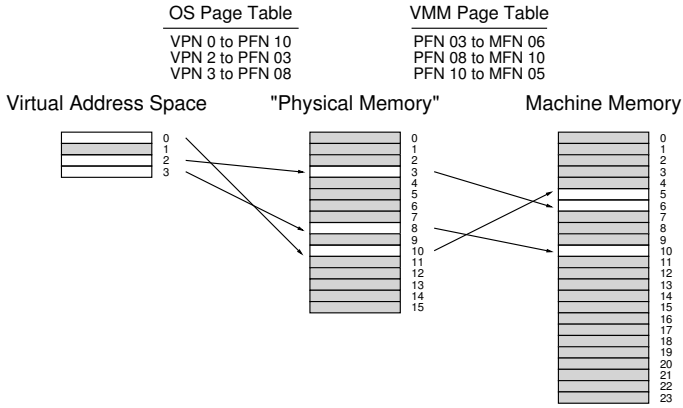


Figure B.1: **VMM Memory Virtualization**

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.

10

Multiprocessor Scheduling (Advanced)

This chapter will introduce the basics of **multiprocessor scheduling**. As this topic is relatively advanced, it may be best to cover it *after* you have studied the topic of concurrency in some detail (i.e., the second major "easy piece" of the book).

After years of existence only in the high-end of the computing spectrum, **multiprocessor** systems are increasingly commonplace, and have found their way into desktop machines, laptops, and even mobile devices. The rise of the **multicore** processor, in which multiple CPU cores are packed onto a single chip, is the source of this proliferation; these chips have become popular as computer architects have had a difficult time making a single CPU much faster without using (way) too much power. And thus we all now have a few CPUs available to us, which is a good thing, right?

Of course, there are many difficulties that arise with the arrival of more than a single CPU. A primary one is that a typical application (i.e., some C program you wrote) only uses a single CPU; adding more CPUs does not make that single application run faster. To remedy this problem, you'll have to rewrite your application to run in **parallel**, perhaps using **threads** (as discussed in great detail in the second piece of this book). Multi-threaded applications can spread work across multiple CPUs and thus run faster when given more CPU resources.

ASIDE: **ADVANCED CHAPTERS**

Advanced chapters require material from a broad swath of the book to truly understand, while logically fitting into a section that is earlier than said set of prerequisite materials. For example, this chapter on multiprocessor scheduling makes much more sense if you've first read the middle piece on concurrency; however, it logically fits into the part of the book on virtualization (generally) and CPU scheduling (specifically). Thus, it is recommended such chapters be covered out of order; in this case, after the second piece of the book.

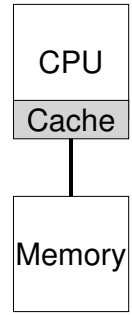


Figure 10.1: **Single CPU With Cache**

Beyond applications, a new problem that arises for the operating system is (not surprisingly!) that of **multiprocessor scheduling**. Thus far we've discussed a number of principles behind single-processor scheduling; how can we extend those ideas to work on multiple CPUs? What new problems must we overcome? And thus, our problem:

CRUX: HOW TO SCHEDULE JOBS ON MULTIPLE CPUS
How should the OS schedule jobs on multiple CPUs? What new problems arise? Do the same old techniques work, or are new ideas required?

10.1 Background: Multiprocessor Architecture

To understand the new issues surrounding multiprocessor scheduling, we have to understand a new and fundamental difference between single-CPU hardware and multi-CPU hardware. This difference centers around the use of hardware **caches** (e.g., Figure 10.1), and exactly how data is shared across multiple processors. We now discuss this issue further, at a high level. Details are available elsewhere [CSG99], in particular in an upper-level or perhaps graduate computer architecture course.

In a system with a single CPU, there are a hierarchy of **hardware caches** that in general help the processor run programs faster. Caches are small, fast memories that (in general) hold copies of *popular* data that is found in the main memory of the system. Main memory, in contrast, holds *all* of the data, but access to this larger memory is slower. By keeping frequently accessed data in a cache, the system can make the large, slow memory appear to be a fast one.

As an example, consider a program that issues an explicit load instruction to fetch a value from memory, and a simple system with only a single CPU; the CPU has a small cache (say 64 KB) and a large main memory.

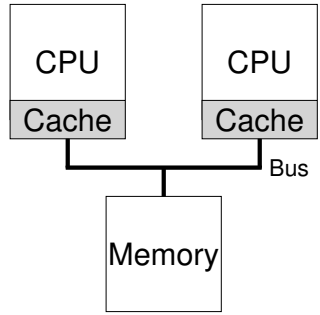


Figure 10.2: **Two CPUs With Caches Sharing Memory**

The first time a program issues this load, the data resides in main memory, and thus takes a long time to fetch (perhaps in the tens of nanoseconds, or even hundreds). The processor, anticipating that the data may be reused, puts a copy of the loaded data into the CPU cache. If the program later fetches this same data item again, the CPU first checks for it in the cache; if it finds it there, the data is fetched much more quickly (say, just a few nanoseconds), and thus the program runs faster.

Caches are thus based on the notion of **locality**, of which there are two kinds: **temporal locality** and **spatial locality**. The idea behind temporal locality is that when a piece of data is accessed, it is likely to be accessed again in the near future; imagine variables or even instructions themselves being accessed over and over again in a loop. The idea behind spatial locality is that if a program accesses a data item at address x, it is likely to access data items near x as well; here, think of a program streaming through an array, or instructions being executed one after the other. Because locality of these types exist in many programs, hardware systems can make good guesses about which data to put in a cache and thus work well.

Now for the tricky part: what happens when you have multiple processors in a single system, with a single shared main memory, as we see in Figure 10.2?

As it turns out, caching with multiple CPUs is much more complicated. Imagine, for example, that a program running on CPU 1 reads a data item (with value D) at address A; because the data is not in the cache on CPU 1, the system fetches it from main memory, and gets the value D. The program then modifies the value at address A, just updating its cache with the new value D'; writing the data through all the way to main memory is slow, so the system will (usually) do that later. Then assume the OS decides to stop running the program and move it to CPU 2. The program then re-reads the value at address A; there is no such data

CPU 2's cache, and thus the system fetches the value from main memory, and gets the old value D instead of the correct value D'. Oops!

This general problem is called the problem of **cache coherence**, and there is a vast research literature that describes many different subtleties involved with solving the problem [SHW11]. Here, we will skip all of the nuance and make some major points; take a computer architecture class (or three) to learn more.

The basic solution is provided by the hardware: by monitoring memory accesses, hardware can ensure that basically the "right thing" happens and that the view of a single shared memory is preserved. One way to do this on a bus-based system (as described above) is to use an old technique known as **bus snooping** [G83]; each cache pays attention to memory updates by observing the bus that connects them to main memory. When a CPU then sees an update for a data item it holds in its cache, it will notice the change and either **invalidate** its copy (i.e., remove it from its own cache) or **update** it (i.e., put the new value into its cache too). Write-back caches, as hinted at above, make this more complicated (because the write to main memory isn't visible until later), but you can imagine how the basic scheme might work.

10.2 Don't Forget Synchronization

Given that the caches do all of this work to provide coherence, do programs (or the OS itself) have to worry about anything when they access shared data? The answer, unfortunately, is yes, and is documented in great detail in the second piece of this book on the topic of concurrency. While we won't get into the details here, we'll sketch/review some of the basic ideas here (assuming you're familiar with concurrency).

When accessing (and in particular, updating) shared data items or structures across CPUs, mutual exclusion primitives (such as locks) should likely be used to guarantee correctness (other approaches, such as building **lock-free** data structures, are complex and only used on occasion; see the chapter on deadlock in the piece on concurrency for details). For example, assume we have a shared queue being accessed on multiple CPUs concurrently. Without locks, adding or removing elements from the queue concurrently will not work as expected, even with the underlying coherence protocols; one needs locks to atomically update the data structure to its new state.

To make this more concrete, imagine this code sequence, which is used to remove an element from a shared linked list, as we see in Figure 10.3. Imagine if threads on two CPUs enter this routine at the same time. If Thread 1 executes the first line, it will have the current value of `head` stored in its `tmp` variable; if Thread 2 then executes the first line as well, it also will have the same value of `head` stored in its own private `tmp` variable (`tmp` is allocated on the stack, and thus each thread will have its own private storage for it). Thus, instead of each thread removing an element from the head of the list, each thread will try to remove the

```
1  typedef struct __Node_t {
2      int              value;
3      struct __Node_t *next;
4  } Node_t;
5
6  int List_Pop() {
7      Node_t *tmp = head;         // remember old head ...
8      int value   = head->value;  // ... and its value
9      head        = head->next;   // advance head to next pointer
10     free(tmp);                  // free old head
11     return value;               // return value at head
12 }
```

Figure 10.3: **Simple List Delete Code**

same head element, leading to all sorts of problems (such as an attempted double free of the head element at line 4, as well as potentially returning the same data value twice).

The solution, of course, is to make such routines correct via **locking**. In this case, allocating a simple mutex (e.g., `pthread_mutex_t m;`) and then adding a `lock(&m)` at the beginning of the routine and an `unlock(&m)` at the end will solve the problem, ensuring that the code will execute as desired. Unfortunately, as we will see, such an approach is not without problems, in particular with regards to performance. Specifically, as the number of CPUs grows, access to a synchronized shared data structure becomes quite slow.

10.3 One Final Issue: Cache Affinity

One final issue arises in building a multiprocessor cache scheduler, known as **cache affinity**. This notion is simple: a process, when run on a particular CPU, builds up a fair bit of state in the caches (and TLBs) of the CPU. The next time the process runs, it is often advantageous to run it on the same CPU, as it will run faster if some of its state is already present in the caches on that CPU. If, instead, one runs a process on a different CPU each time, the performance of the process will be worse, as it will have to reload the state each time it runs (note it will run correctly on a different CPU thanks to the cache coherence protocols of the hardware). Thus, a multiprocessor scheduler should consider cache affinity when making its scheduling decisions, perhaps preferring to keep a process on the same CPU if at all possible.

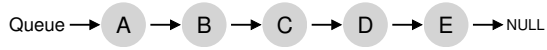
10.4 Single-Queue Scheduling

With this background in place, we now discuss how to build a scheduler for a multiprocessor system. The most basic approach is to simply reuse the basic framework for single processor scheduling, by putting all jobs that need to be scheduled into a single queue; we call this **single-queue multiprocessor scheduling** or **SQMS** for short. This approach has the advantage of simplicity; it does not require much work to take an existing policy that picks the best job to run next and adapt it to work on more than one CPU (where it might pick the best two jobs to run, if there are two CPUs, for example).

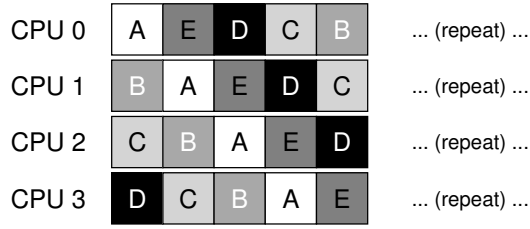
However, SQMS has obvious shortcomings. The first problem is a lack of **scalability**. To ensure the scheduler works correctly on multiple CPUs, the developers will have inserted some form of **locking** into the code, as described above. Locks ensure that when SQMS code accesses the single queue (say, to find the next job to run), the proper outcome arises.

Locks, unfortunately, can greatly reduce performance, particularly as the number of CPUs in the systems grows [A91]. As contention for such a single lock increases, the system spends more and more time in lock overhead and less time doing the work the system should be doing (note: it would be great to include a real measurement of this in here someday).

The second main problem with SQMS is cache affinity. For example, let us assume we have five jobs to run (A, B, C, D, E) and four processors. Our scheduling queue thus looks like this:

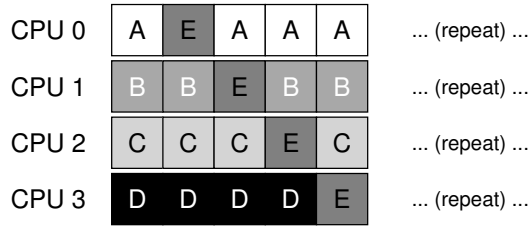


Over time, assuming each job runs for a time slice and then another job is chosen, here is a possible job schedule across CPUs:



Because each CPU simply picks the next job to run from the globally-shared queue, each job ends up bouncing around from CPU to CPU, thus doing exactly the opposite of what would make sense from the standpoint of cache affinity.

To handle this problem, most SQMS schedulers include some kind of affinity mechanism to try to make it more likely that process will continue to run on the same CPU if possible. Specifically, one might provide affinity for some jobs, but move others around to balance load. For example, imagine the same five jobs scheduled as follows:



In this arrangement, jobs A through D are not moved across processors, with only job E **migrating** from CPU to CPU, thus preserving affinity for most. You could then decide to migrate a different job the next time through, thus achieving some kind of affinity fairness as well. Implementing such a scheme, however, can be complex.

Thus, we can see the SQMS approach has its strengths and weaknesses. It is straightforward to implement given an existing single-CPU scheduler, which by definition has only a single queue. However, it does not scale well (due to synchronization overheads), and it does not readily preserve cache affinity.

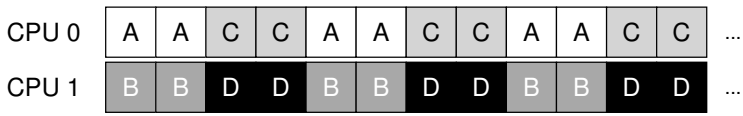
10.5 Multi-Queue Scheduling

Because of the problems caused in single-queue schedulers, some systems opt for multiple queues, e.g., one per CPU. We call this approach **multi-queue multiprocessor scheduling** (or **MQMS**).

In MQMS, our basic scheduling framework consists of multiple scheduling queues. Each queue will likely follow a particular scheduling discipline, such as round robin, though of course any algorithm can be used. When a job enters the system, it is placed on exactly one scheduling queue, according to some heuristic (e.g., random, or picking one with fewer jobs than others). Then it is scheduled essentially independently, thus avoiding the problems of information sharing and synchronization found in the single-queue approach.

For example, assume we have a system where there are just two CPUs (labeled CPU 0 and CPU 1), and some number of jobs enter the system: A, B, C, and D for example. Given that each CPU has a scheduling queue now, the OS has to decide into which queue to place each job. It might do something like this:

Depending on the queue scheduling policy, each CPU now has two jobs to choose from when deciding what should run. For example, with **round robin**, the system might produce a schedule that looks like this:



MQMS has a distinct advantage of SQMS in that it should be inherently more scalable. As the number of CPUs grows, so too does the number of queues, and thus lock and cache contention should not become a central problem. In addition, MQMS intrinsically provides cache affinity;

jobs stay on the same CPU and thus reap the advantage of reusing cached contents therein.

But, if you've been paying attention, you might see that we have a new problem, which is fundamental in the multi-queue based approach: **load imbalance**. Let's assume we have the same set up as above (four jobs, two CPUs), but then one of the jobs (say C) finishes. We now have the following scheduling queues:

If we then run our round-robin policy on each queue of the system, we will see this resulting schedule:

As you can see from this diagram, A gets twice as much CPU as B and D, which is not the desired outcome. Even worse, let's imagine that both A and C finish, leaving just jobs B and D in the system. The scheduling queues will look like this:

As a result, CPU 0 will be left idle! *(insert dramatic and sinister music here)* And hence our CPU usage timeline looks sad:

So what should a poor multi-queue multiprocessor scheduler do? How can we overcome the insidious problem of load imbalance and defeat the evil forces of ... the Decepticons[1]? How do we stop asking questions that are hardly relevant to this otherwise wonderful book?

[1] Little known fact is that the home planet of Cybertron was destroyed by bad CPU scheduling decisions. And now let that be the first and last reference to Transformers in this book, for which we sincerely apologize.

CRUX: HOW TO DEAL WITH LOAD IMBALANCE
How should a multi-queue multiprocessor scheduler handle load imbalance, so as to better achieve its desired scheduling goals?

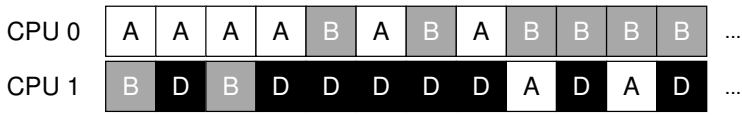
The obvious answer to this query is to move jobs around, a technique which we (once again) refer to as **migration**. By migrating a job from one CPU to another, true load balance can be achieved.

Let's look at a couple of examples to add some clarity. Once again, we have a situation where one CPU is idle and the other has some jobs.

In this case, the desired migration is easy to understand: the OS should simply move one of B or D to CPU 0. The result of this single job migration is evenly balanced load and everyone is happy.

A more tricky case arises in our earlier example, where A was left alone on CPU 0 and B and D were alternating on CPU 1:

In this case, a single migration does not solve the problem. What would you do in this case? The answer, alas, is continuous migration of one or more jobs. One possible solution is to keep switching jobs, as we see in the following timeline. In the figure, first A is alone on CPU 0, and B and D alternate on CPU 1. After a few time slices, B is moved to compete with A on CPU 0, while D enjoys a few time slices alone on CPU 1. And thus load is balanced:



Of course, many other possible migration patterns exist. But now for the tricky part: how should the system decide to enact such a migration?

One basic approach is to use a technique known as **work stealing** [FLR98]. With a work-stealing approach, a (source) queue that is low on jobs will occasionally peek at another (target) queue, to see how full it is. If the target queue is (notably) more full than the source queue, the source will "steal" one or more jobs from the target to help balance load.

Of course, there is a natural tension in such an approach. If you look around at other queues too often, you will suffer from high overhead and have trouble scaling, which was the entire purpose of implementing

the multiple queue scheduling in the first place! If, on the other hand, you don't look at other queues very often, you are in danger of suffering from severe load imbalances. Finding the right threshold remains, as is common in system policy design, a black art.

10.6 Linux Multiprocessor Schedulers

Interestingly, in the Linux community, no common solution has approached to building a multiprocessor scheduler. Over time, three different schedulers arose: the O(1) scheduler, the Completely Fair Scheduler (CFS), and the BF Scheduler (BFS)[2]. See Meehean's dissertation for an excellent overview of the strengths and weaknesses of said schedulers [M11]; here we just summarize a few of the basics.

Both O(1) and CFS use multiple queues, whereas BFS uses a single queue, showing that both approaches can be successful. Of course, there are many other details which separate these schedulers. For example, the O(1) scheduler is a priority-based scheduler (similar to the MLFQ discussed before), changing a process's priority over time and then scheduling those with highest priority in order to meet various scheduling objectives; interactivity is a particular focus. CFS, in contrast, is a deterministic proportional-share approach (more like Stride scheduling, as discussed earlier). BFS, the only single-queue approach among the three, is also proportional-share, but based on a more complicated scheme known as Earliest Eligible Virtual Deadline First (EEVDF) [SA96]. Read more about these modern algorithms on your own; you should be able to understand how they work now!

10.7 Summary

We have seen various approaches to multiprocessor scheduling. The single-queue approach (SQMS) is rather straightforward to build and balances load well but inherently has difficulty with scaling to many processors and cache affinity. The multiple-queue approach (MQMS) scales better and handles cache affinity well, but has trouble with load imbalance and is more complicated. Whichever approach you take, there is no simple answer: building a general purpose scheduler remains a daunting task, as small code changes can lead to large behavioral differences. Only undertake such an exercise if you know exactly what you are doing, or, at least, are getting paid a large amount of money to do so.

[2] Look up what BF stands for on your own; be forewarned, it is not for the faint of heart.

References

[A90] "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors"
Thomas E. Anderson
IEEE TPDS Volume 1:1, January 1990
A classic paper on how different locking alternatives do and don't scale. By Tom Anderson, very well known researcher in both systems and networking. And author of a very fine OS textbook, we must say.

[B+10] "An Analysis of Linux Scalability to Many Cores Abstract"
Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, Nickolai Zeldovich
OSDI '10, Vancouver, Canada, October 2010
A terrific modern paper on the difficulties of scaling Linux to many cores.

[CSG99] "Parallel Computer Architecture: A Hardware/Software Approach"
David E. Culler, Jaswinder Pal Singh, and Anoop Gupta
Morgan Kaufmann, 1999
A treasure filled with details about parallel machines and algorithms. As Mark Hill humorously observes on the jacket, the book contains more information than most research papers.

[FLR98] "The Implementation of the Cilk-5 Multithreaded Language"
Matteo Frigo, Charles E. Leiserson, Keith Randall
PLDI '98, Montreal, Canada, June 1998
Cilk is a lightweight language and runtime for writing parallel programs, and an excellent example of the work-stealing paradigm.

[G83] "Using Cache Memory To Reduce Processor-Memory Traffic"
James R. Goodman
ISCA '83, Stockholm, Sweden, June 1983
The pioneering paper on how to use bus snooping, i.e., paying attention to requests you see on the bus, to build a cache coherence protocol. Goodman's research over many years at Wisconsin is full of cleverness, this being but one example.

[M11] "Towards Transparent CPU Scheduling"
Joseph T. Meehean
Doctoral Dissertation at University of Wisconsin—Madison, 2011
A dissertation that covers a lot of the details of how modern Linux multiprocessor scheduling works. Pretty awesome! But, as co-advisors of Joe's, we may be a bit biased here.

[SHW11] "A Primer on Memory Consistency and Cache Coherence"
Daniel J. Sorin, Mark D. Hill, and David A. Wood
Synthesis Lectures in Computer Architecture
Morgan and Claypool Publishers, May 2011
A definitive overview of memory consistency and multiprocessor caching. Required reading for anyone who likes to know way too much about a given topic.

[SA96] "Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation"
Ion Stoica and Hussein Abdel-Wahab
Technical Report TR-95-22, Old Dominion University, 1996
A tech report on this cool scheduling idea, from Ion Stoica, now a professor at U.C. Berkeley and world expert in networking, distributed systems, and many other things.

Multi-Processor Systems

Much of the discussion in this course has considered the operating system to be running on a time-shared uni-processor ... and this perspective is adequate to fully understand most of those topics. But increasingly many modern computer systems are now multi-processor:

> Multiple general purpose CPUs (as opposed to GPUs) that are capable of running unrelated programs or threads (unlike SIMD array processors) and (to some degree) share memory and I/O devices.

These systems are interesting because they are independent enough to encounter many of the problems associated with distributed computing ... but (because they share memory and I/O devices) do things that push the distributed systems envelope. As people develop applications to exploit these platforms, it is important that they understand the issues they present.

Why Build Multi-Processor Systems

We continue to find applications that require ever more computing power. Sometimes these problems can be solved by horizontally scaled systems (e.g. thousands of web servers). But some problems demand, not more computers, but faster computers. Consider a single huge database, that each year, must handle twice as many operations as it served the previous year. Distributed locking, for so many parallel transactions on a single database, could be prohibitively expensive. The (seemingly also prohibitively expensive) alternative would be to buy a bigger computer every year.

Long ago it was possible to make computers faster by shrinking the gates, speeding up the clock, and improving the cooling. But eventually we reach a point of dimmisnishing returns where physics (the speed of light, information theory, thermodynamics) makes it ever more difficult to build faster CPUs. Recently, most of our improvements in processing speed have come from:

- smarter pipe-lining and increasingly parallel and speculative execution
- putting more cores per chip, more chips per board, and more boards per computer system.

But it is reasonable to ask whether or not 16x3B instructions per second is actually equivalent to 48B instructions per second? The answer (see Amdahl's Law) depends on whether or not your application can be divided into 16 or more parallely executable sub-tasks. Fortunately, modern operating systems tend to run large numbers of processes, and expensive computations are increasingly designed to be executable in multiple parallel threads..

For these reasons, multi-processor is the dominant architecture for powerful servers and desktops. And, as the dominant architecture, operating systems must do a good job of exploiting them.

Multi-Processor Hardware

The above general defintion covers a wide range of architectures, that actually have very different characteristics. And so it is useful, to overview the most prominent architectures.

Hyper-Threading

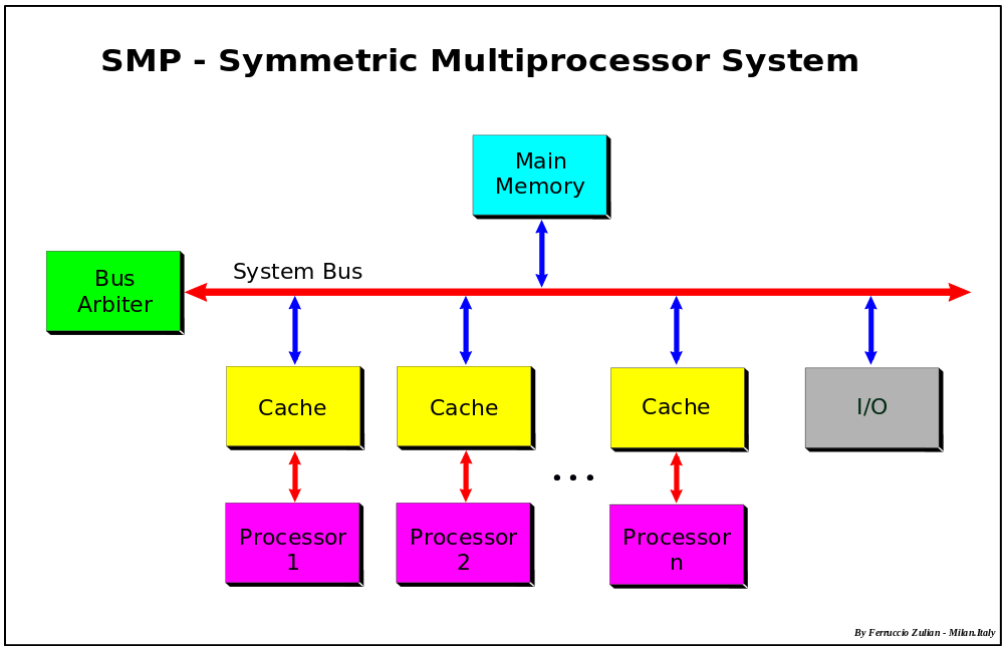
CPUs are much faster than memory. A 2.5GHz CPU might be able to execute more than 5 Billion instructions for second. Unfortunately, 80ns memory can only deliver 12 Million fetches or stores per second. This is almost a 1000x mis-match in performance. The CPU has multiple levels of cache to ensure that we seldom have to go to memory, but even so, the CPU spends a great deal of time waiting for memory.

The idea of hyper-threading is to give each core two sets of general registers, and the ability to run two independent threads. When one of those threads is blocked (waiting for memory) the other thread can be using the execution engine. Think of this as non-preemptive time-sharing at the micro-code level. It is comon for a pair of hyper-threads to get 1.2-1.8 times the instructions per second that a single thread would have gotten on the same core. It is theoretically possible to get 2x hyper-threading, but a thread might run out of L1 cache for a long time without blocking, or perhaps both hyper-threads are bocked waiting for memory.

From a performance point-of-view, it is important to understand that both hyper-threads are running in the same core, and so sharing the same L1 and L2 cache. Thus hyper-threads that use the same address space will exhibit better locality, and hence run much better than hyper-threads that use different address spaces.

Symmetric Multi-Processors

A Symmetric Multi-Processor has some number of cores, all connected to the same memory and I/O busses. Unlike hyper-threads these cores are completely independent execution engines, and (modulo limitations on memory and bus throughput) N cores should be able to execute N times as many instructions per second as a single core.

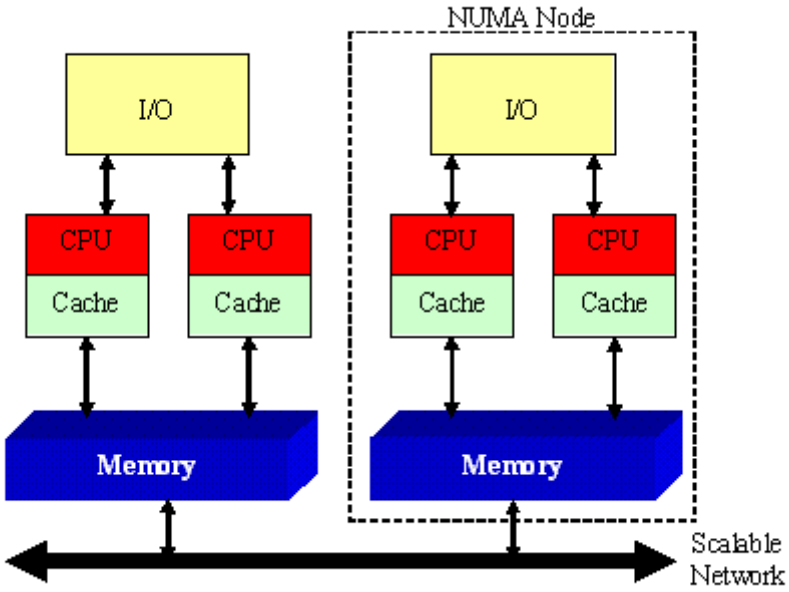


Cache Coherence

As mentioned previously, much of processor performance is a result of caching. In most SMP systems, each processor has its own L1/L2 caches. This creates a potential *cache-coherency* problem if (for instance) processor 1 updates a memory location whose contents have been cached by processor 2. Program execution based on stale cache entries would result in incorrect results, and so must be prevented. There are a few general approaches to maintaining cache coherency (ensuring that there are no disagreements about the current contents of any cache line), and most SMP systems with per-processor caching incorporate some Cache Coherency Mechanism to address this issue.

Cache Coherent Non-Uniform Memory Architectures

It is not feasible to create fast memory controllers that can provide concurrent access to large numbers of cores, and eventually memory bandwidth becomes the bottleneck that prevents scaling to larger numbers of CPUs. A Non-Uniform Memory Architecture addresses this problem by giving each node or CPU its own high-speed local memory, and interconnecting all of the memory busses with a slower but more scalable network.



Operations to local memory may be several times faster than operations to remote memory, and the maximum throughput of the scalable network may be a small fraction of the per-node local memory bandwidth. Such an architecture might provide nearly linear scaling to much larger numbers of processors, but only if we can ensure that most of the memory references are local. The Operating System might be able to deal with the different memory access speeds by trying allocate memory for each process from the CPU on which that process is running. But there will still be situations where multiple CPUs need to access the same memory. To ensure correct execution, we must maintain coherency between all of the per-node/per-CPU caches. This means that, in addition to servicing remote memory read and write requests, the scalable network that interconnects the nodes must also provide cache coherency. Such architectures are called *Cache Coherent Non-Uniform Memory Architectures* (CC-NUMA), and the implementing networks are called *Scalable Coherent Interconnects*. The best known Scalable Coherent Interconnects are probably Intel's Quick Path Interconnect (QPI), and AMD's HyperTransport.

Power Management

The memory and cache interconnections are probably the most interesting part of a multi-processor system, but power management is another very important feature. A multi-core system can consume a huge amount of power ... and most of the time it does not need most of the cores. Many multi-processor systems include mechanisms to slow (or stop) the clocks on unneeded cores, which dramatically reduces system power consumption. This is not a slow process, like a sleep and reboot. A core can be returned to full speed very quickly.

Multi-Processor Operating Systems

To exploit a multi-processor system, the operating system must be able to concurrently manage multiple threads/processes on each of the available CPU cores. One of the earliest approaches was to run the operating system on one core, and applications on all of the others. This works reasonably for a small number of cores, but as the number of cores increases, the OS becomes the primary throughput bottleneck. Scaling to larger numbers of cores requires the operating system itself to run on multiple cores. Running efficiently on multiple cores requires the operating system to carefully choose which threads/processes to run on which cores and what resources to allocate to them.

When we looked at distributed systems, we saw (e.g. Deutsch's Seven Falacies) that the mere fact that a network is capable of distributing every operation to an arbitrary node does not make doing so a good idea. It will be seen that the same caveat applies to multi-processor systems.

Scheduling

If there are threads (or processes) to run, we would like to keep all of the cores busy. If there are not threads (or processes) to run, we would like to put as many cores as possible into low power mode. It is tempting to think that we can just run each thread/process on the next core that becomes available (due to a process blocking or getting a time-slice-end). But some cores may be able to run some threads (or processes) far more efficiently than others.

- dispatching a thread from the same process as the previous thread (to occupy that core) may be much less expensive because re-loading the page table (and flushing all cached TLB entries) is a very expensive operation.
- a thread in the same process may run more efficiently because shared code and data may exploit already existing L1/L2 cache entries.
- threads that are designed to run concurrently (e.g. parallel producer and consumer communicating through shared memory) should be run on distinct cores.

Thus, the choice of when to run which thread in which core is not an arbitrary one. The scheduler must consider what process was last running in each core. It may make more sense to leave one core idle, and delay executing some thread until its preferred core becomes available. The operating system will try to make intelligent decisions, but if the developers understand how work can best be allocated among multi-processor cores, they can advise the operating system with operations like *sched_setaffinity(2)* and *pthead_setaffinity_np(3)*.

Synchronization

Sharing data between processes is relatively rare in user mode code. But the operating system is full of shared data (process table entries, file descriptors, scheduling and I/O queues, etc). In a uni-processor, the primary causes of race conditions are preemptive scheduling and I/O interrupts. Both of these problems can be managed by disabling (selected) interrupts while executing critical sections ... and many operating systems simply declare that preemptions cannot occur while executing in the operating system.

These techniques cease to be effective once the operating system is running on multiple CPUs in a multi-processor system. Disabling interrupts cannot prevent another core from performing operations on a single global object (e.g. I-node). Thus multi-processor operating systems require some other means to ensure the integrity global data structures. Early multi-processor operating systems tried to create a single, global, kernel lock ... to ensure that only one process at a time could be executing in the operating system. But this is essentially equivalent to running the operating system on a single CPU. As the number of cores increases, the (single threaded) operating system becomes the scalability bottleneck.

The Solution to this problem is finer grained locking. And as the number of cores increased, the granularity required to achieve high parallelism became ever finer. Depending on the particular shared resource and operations, different synchronizations may have to be achieved with different mechanisms (e.g. compare and swap, spin-locks, interrupt disables, try-locks, or blocking mutexes). Moreover for every resource (and combination of resources) we need a plan to prevent deadlock. Changing complex code that involves related updates to numerous data structures to implement fine-grained locking is difficult to do (often requiring significant design changes) and relatively brittle (as maintainers make changes that fail to honor all of the complex synchronization rules). If a decision is made to transition the operating system to finer grained locking, it becomes much more difficult for third party developers to build add-ons (e.g. device drivers and file systems) that will work with the finer grained locking schemes in the hosting operating system.

Because of this complexity, there are relatively few operating systems that are able to efficiently scale to large numbers of multi-processor cores. Most operating systems have chosen simplicity and maintainability over scalability.

Device I/O

If an I/O operation is to be initiated, does it matter which CPU initiates it? When an I/O interrupt comes in to a multi-processor system, which processor should be interrupted? There are a few reasons we might want to choose carefully which cores handle which I/O operations:

- as with scheduling, sending all operations for a particular device to a particular core may result in more L1/L2 cache hits and more efficient execution.
- syncrhonization between the synchronous (resulting from system calls) and asynchronous (resulting from interrupts) portions of a device driver if they are all executing in the same CPU.
- each CPU has a limited I/O throughput, and we may want to balance activity among the available cores.
- some CPUs may be bus-wise closer to some I/O devices, so that operations go more quickly initiated from some cores.

Many multi-processor architectures have interrupt controllers that are configurable for which interrupts should be delivered to which processors.

Non-Uniform Memory Architectures

CC-NUMA is only viable if we can ensure that the vast majority of all memory references can be satisfied from local memory. Doing this turns out to create a lot of complexity for the operating system.

When we were discussing uni-processor memory allocation, we observed that significant savings could be achieved if we shared a single copy of a (read only) load module among all processes that were running that program. This ceases to be true when those processes are running on distinct NUMA nodes. Code and other read-only data should be have a separate copy (in local memory) on each NUMA node. The cost (in terms of wasted memory) is neglibile in comparison performance gains from making all code references local.

When a program calls *fork(2)* to create a new process, *exec(2)* to run a new program, or *sbrk(2)* to expand its address space, the required memory should always be allocated from the node-local memory pool. This creates a very strong affinity between processes and NUMA nodes. If it is necessary to migrate a process to a new NUMA node, all of its allocated code and data segments should be copied into local memory on the target node.

As noted above, the operating system is full of data structures that are shared among many cores. How can we reduce the number or cost of remote memory references associated with those shared data structures? If the updates are few, sparse and random, there may be little we can do to optimize them ... but their costs will not be high. When more intensive use of shared data structures is required, there are two general appoaches:

1. move the data to the computation
    - lock the data structure.
    - copy it into local memory.
    - update the global pointer to reflect its new location.
    - free the old (remote) copy.
    - perform all subsequent operations on the (now) local copy.
2. move the computation to the data
    - look up the node that owns the resource in question.
    - send a message requesting it to perform the required operations.
    - await a response.

In practice, both of these techniques are used ... the choice determined by the particulars of the resource and its access.

As with fine-grained synchronization of kernel data structures, this turns out to be extremely complicated. Relatively few operating systems have been willing to pay this cost, and so (again) most opt for simplicity and maintainability over performance and scalability.

Eventual Consistency

Wednesday, November 30, 2016 4:32 PM

Eventual consistency

Eventual consistency is a consistency model used in distributed computing to achieve high availability that informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.[1] Eventual consistency is widely deployed in distributed systems, often under the moniker of optimistic replication,[2] and has origins in early mobile computing projects.[3] A system that has achieved eventual consistency is often said to have **converged**, or achieved **replica convergence**.[4] Eventual consistency is a weak guarantee – most stronger models, like linearizability are trivially eventually consistent, but a system that is merely eventually consistent does not usually fulfill these stronger constraints.

Eventually consistent services are often classified as providing BASE (**B**asically **A**vailable, **S**oft state, **E**ventual consistency) semantics, in contrast to traditional ACID (**A**tomicity, **C**onsistency, **I**solation, **D**urability) guarantees.[5][6] Eventual consistency is sometimes criticized[7] as increasing the complexity of distributed software applications. This is partly because eventual consistency is purely a liveness guarantee (reads eventually return the same value) and does not make safety guarantees: an eventually consistent system can return any value before it converges.

Conflict resolution

In order to ensure replica convergence, a system must reconcile differences between multiple copies of distributed data. This consists of two parts:

- exchanging versions or updates of data between servers (often known as **anti-entropy**);[8] and
- choosing an appropriate final state when concurrent updates have occurred, called **reconciliation**.

The most appropriate approach to reconciliation depends on the application. A widespread approach is *"last writer wins"*.[1] Another is to invoke a user-specified conflict handler.[4] Timestamps and vector

clocks are often used to detect concurrency between updates.

Reconciliation of concurrent writes must occur sometime before the next read, and can be scheduled at different instants:[3][9]

- Read repair: The correction is done when a read finds an inconsistency. This slows down the read operation.
- Write repair: The correction takes place during a write operation, if an inconsistency has been found, slowing down the write operation.
- Asynchronous repair: The correction is not part of a read or write operation.

Strong eventual consistency

Whereas eventual consistency is only a liveness guarantee (updates will be observed eventually), **strong eventual consistency** (SEC) adds the safety guarantee that any two nodes that have received the same (unordered) set of updates will be in the same state. If, furthermore, the system is monotonic, the application will never suffer rollbacks. Conflict-free replicated data types are a common approach to ensuring SEC.[10]

Cluster Concepts

by Mark Kampe

Cluster

There are many different types of clusters. Common types include:

- load sharing clusters, which divide work among the members.
- high availability clusters, where back-up nodes take over when primary nodes fail.
- information sharing clusters, which ensure the dissemination of information throughout a network.

There are so many different goals and architectures that Greg Phister (in "In Search of Clusters") came to the conclusion that it is very difficult to even define the term. About the only thing we can say for certain is that a cluster is a networked connection of nodes, all of whom agree that they are part of a cluster.

Membership

If a cluster is defined as a networked connection of nodes who consider themselves to be participants in the cluster, then obviously "membership" is a key concept.

We can distinguish two types of membership:

- potential, eligible or designated members
- active or currently participating members

This distinction is important because only active members can communicate with one-another. Thus it is that the term "membership" is most commonly used to describe only the currently participating members.

It is very important to know who the current (active) cluster members are. We may, for instance, be required to make sure that each of them has been informed of some operation before we are allowed to perform it. Cluster membership often comes with responsibilities (e.g. a guarantee to respond to certain requests within a certain period if time). Thus it is vital that we know when nodes enter and leave the cluster.

In most clusters, a node has to be explicitly configured or provisioned into the cluster ... so that the set of potential members is well known, and perhaps even closed to new members. There are, however, some types of clusters where any node is welcooe to join at any time.

Node Redundancy

In a clustered system, work is divided among the active members. To reduce distributed synchronization, it is common to partition the work (e.g. desigate each server responsible for a certain subset (e.g. a file system, a range of keys, etc) of requests, and route all requests to their designated owner). In such systems, we can talk about *primaries* (the designated owners) and *secondaries* (nodes who are prepared to take over for a primary if he fails).

There are three fundamentally different approaches to take to high availability:

- Active/Stand-By
  The system is divided into *active* and *stand-by* nodes. The incoming requests are partitioned among the active nodes. The stand-by nodes are idle until an active node fails, at which point a stand-by node takes over his work.

- Active/Active
  The incoming requests are partitioned among all of the available nodes. If one node fails, his work will be redistributed among the survivors.

An active/active architecture achieves better resource utilization, and so may be more cost-effective. But when a failure occurrs, the load on the surviving nodes is increased and so performance may suffer. An active/stand-by architecture normally has idle capacity, but may not suffer any performance degradataion after a failure.

We can also look at how quickly a successor is able to take over a failed node's responsibilities. In some architectures, all operations are mirrored to the secondaries, enabling them to very quickly assume the primary role. Such secondaries are called *hot standbys*. In other systems, the secondary waits to be notified of the primary's failure, after which it opens and reads the failed primary's last check-point or journal. The former approach results in more network and CPU load, but much faster fail-overs. The latter approach consumes fewer resources during normal operation, but may take much longer to resume service after a primary has failed.

Heart Beat

Ideally nodes will announce the fact that they are joining the cluster, or are about to leave it. This is not always the case:

1. a system may crash.
2. the clustering applications may crash.
3. a node may become so busy that the clustering applications cannot run.
4. a network interface or link may fail.

Since we cannot be sure that member will notify the other members before he leaves the cluster, we need a means of detecting a member who has dropped unexpectedly out of the cluster. The most common techique is called a "heart beat". A heart beat is a message, regularly exchanged between cluster members. These may be special messages, or they may be piggy-backed on other traffic. If too much time passes without our having received a heart-beat from some node, we will assume that node has failed, and is no longer an active cluster member.

The failure of a node may (in some clusters) have serious consequences (e.g. the freeing of all resources allocated to that node, and the aborting of all in progress transactions from that node). To prevent "false alarms", many systems perform heart-beats over multiple channels, or have a back-up link with which they attempt to confirm a failure before reporting a node to be dead.

Cluster Master and Election

It is often convenient to elect or designate one node to be the cluster master:

- Coming to a mutual agreement between multiple nodes can be a complex process (e.g. Three Phase Commits). If one node is designated a cluster master, that node can serve as a central point of syncrhronization and/or control for operations in the cluster.
- Rather than requiring all nodes to heart-beat one-another, it is more economical to simply have all nodes heart-beat with the cluster master. He will detect the failure of any other node, and all nodes will detect his failure.

The election of a cluster master may, itself, be a complex process ... but having performed that process may eliminate the need for any further negotiations. There are numerous well established election/concensus algorithms. One of the best known is Leslie Lamport's Paxos algorithm.

Split Brain

A pathological network failure might divide a cluster into multiple sub-clusters, which cannot communicate with one-another. Such an event is sometimes referred to as a "partitioning" of the network. If the cluster manages critical resources (e.g. a database or nuclear warhead), it is possible that the independent sub-clusters will all continue operating and make independent but incompatible decisions. Such a condition is called "split-brain" (as if two halves of our brain were working independently and at cross-purposes).

There are two standard approaches to preventing "split-brain":

a. quorum
b. voting devices

Quorum

If there are N potential members in a cluster, we can build in a rule that says a cluster cannot form with fewer than (N/2)+1 members. This acomplishes two purposes:

- It makes it impossible for any partitioning to result in two viable sub-clusters (because N nodes cannot be divided into two groups that both contain at least (N/2)+1 nodes).
- It ensures that any decision made (and persisted) by this quorum will be remembered by any future quorum (because any group of (N/2)+1 nodes will have at least one member in common with every other group of (N/2)+1 nodes that has ever existed.

The problem with using a numerical quorum is that if (N/2)+1 nodes have been damaged, it will be impossible for the surviving nodes to form a new cluster ... even if there is no split-brain.

Voting Devices

If there is a single piece of hardware in the cluster, that must be present for the cluster to function, and that can only be owned by one node, that device can be used as a voting device.

Consider, for instance, a shared disk. If that disk is absolutely required to provide service, a node that no longer has access to that disk cannot provide service (and hence is not eligible to form a cluster). But what if two nodes can both talk to the disk, but cannot communicate with one-another? They may be able to use the disk as a voting device ... e.g. by writing a recent time-stamp into a well-known block.

Some clusters include resources that can easily serve as voting devices. There are also specially built (very reliable) voting devices that exist solely for this purpose. If there is a voting device, a cluster could be formed by a single node ... because the voting device would prevent split-brain.

Fencing

What if, you were not only sufferring from schizophrenia, but the other side of your brain had actually gone rogue, and was trying to commit acts of mayhem against you and others? In some clustered systems, it must be assumed that if a node has fallen out of the cluster, he is no longer trust-worthy ... and must be "fenced-out" of the cluster. There are two common approaches to fencing:

- reservable devices
  Some devices can be told which interface to listen to, and not to listen to the other interface. This is often done with dual-ported disks. The node that has seized the quorum device will then instruct the quorum device not to accept commands from any other node.
- remote power control
  Some clustered systems come with remote power controllers, and a node that has seized control of the

cluster from a previous (apparently failed) cluster master will often power-off or reset the previous master, to ensure that he does not continue to vie for control of the cluster and its resources.

Horizontally Scalable Systems

The term *horizontally scalable* refers to systems whose capacity and throughput are increased by adding additional nodes. This is in distinction to *vertically scaled* systems, where adding capacity and throughput generally involves replacing smaller nodes with larger and more powerful ones.

It seems that most technological advancements involve building increasingly complex systems that adapt to increasingly subtle inputs. But we should also recognize that improved understandings and technologies often enable us to build systems that are simultaneously superior to, and in many ways, much simpler than their predecessors. Horizontally scaled systems are based on protocols developed for client-server distributed storage and distributed computing systems. But, unlike their more complex cousins, their evolution has guided by the principles of maximum independence (loose coupling) and and parallelism.

Not all problems can be solved with a horizontally scalable architecture. But where they work, they work very well. This makes horizontally scaled architectures well-worthy of study.

Goals of Horizontally Scalable Systems

These systems started with many of the same goals shared by most distributed systems: scalability of capacity and throughput, reliability, and availability. But whereas earlier distributed systems attempted to provide single-system services and semantics, horizontally scaled systems offer simplified (and decidedly non-Posix) semantics. Most of these simplifications have a few basic (and highly inter-related) goals:

- improve scalability and robustness by eliminating the need for synchronization and communication between parallel servers.
- improve flexibility and performance by enabling the (non-disruptive) addition or removal of servers at any time.
- stateless protocols that permit requests to be arbitrarily distributed and redistributed among those servers.
- turn servers into standardized components, easily deployed and requiring little or no configuration.
- service protocols that are designed to minimize the potential modes of failure and enable simple recoveries.
- service protcols that are optimized for throughput (vs latency) over very large (e.g. continental) distances.
- service protocols that exploit the numerous optimizations and other features that have come to be standard in networks that serve heavy HTTP traffic.

And, as a (not unintentional) side-effect, these systems are much simpler to design, maintain, deploy, and manage than other (more complex) distributed systems.

Horizontally Scalable Hardware Platforms

All services in horizontally scalable systems are exchanged via network messages. A node in such a system is simpy a computer that implements the specified protocols. The instruction set, operating system, memory capacity, and storage technologies are merely implementation details ... of little importance to the system design. In fact, ignoring deployment issues, there should be no problem running a service where every one of the server nodes was a different instruction set, running a different operating system/version, and using different network interfaces and storage controllers.

This high degree of platform independence has made it very common to run horizontally scalable systems on virtual machines. When a new server is needed, we simply clone a standard image, and boot up a new virtual machine. Thirty seconds later, a new server is running. A service run on a virtual machine may be a little less

efficient than it would be if run on physical hardware, but the flexibility and ease of deployment and capacity balancing greatly outweigh any reduced efficiency.

Note, however, that it is not the case that all virtual servers are equivalent to generic desktops. High end VM servers may include GPUs, NVRAM, flash memory, or other special hardware than can be allocated to VMs. With access to such acceleration hardware, a service running in a virtual machine may be able to rival the performance of (much more expensive) custom-built hardware.

If a horizontally scalable system is comprised of many servers, and new servers can be added at any time, it is essential that they all have appropriate network connectivity (for client-facing traffic, back-side servers, and peer-to-perr replication). It is not practical to rack new switches and connect new cables each time a new server is to be integrated into a particular service. Rather, large computing facilities are moving towards *Software Defined Networking* where all servers (virtual or real) are connected to large and versatile switches that can easily be reconfigured to create what ever virtual network topologies and capacities are required for the service to be provided.

Storage is a similar problem. New virtual machines need virtual disks for the operating system to boot off of, and to store the content they will manage. Large computing facilities are also moving towards *Software Defined Storage*, where intelligent storage controllers draw on pools of physical disks (or SSDs) to create virtual disks (LUNs) with the required capacity, redundancy, and performance. The new servers have no idea what the underlying storage is, but merely use remote disk access protocols (e.g. Fibre Chanel or iSCSI) to access configured storage.

Horizontally Scalable System Architectures

The individual servers in a horizontally scaled architecture are highly independent. They do not share CPUs, disks, network interfaces, or a single operating system instance. This means it is unlikely that any single failure would affect multiple servers. Even enclosure components (e.g. fans and power supplies) or networking components (e.g. switches and routers) that might be shared by multiple servers have redundance. Such systems are said to have no *single point of failure* or a *shared nothing architecture*. Both mean that there is no single (shared) component whose failure could affect multiple systems. Such design is key to reliability and availability.

Even if there are no single points of failure in the computing and storage components, all of the servers in a single room (or campus) could potentially be taken down by a regional disaster (power failure, flood, earthquake, etc). For this reason, highly available, and highly reliable systems often make copies and include back-up servers that are far away (e.g. in other cities). Far separated facilities that are unlikely to be affected by a single regional disaster are often referred to as being in distinct *availability zones*. If the servers in one availability zone go down, the work can be handed off to servers in a different availability zone. The ability to fail over to distant copies and serves is sometimes called *geographic disaster recovery*.

Horizontally Scalable Software Architectures

The classic horizontally scaled system is a farm of web-servers, all of which have access to/copies of the same content. As demand increases, more servers are brought on-line. When each server starts up, it is made known to a network switch that starts sending it requests.

- Since HTTP is a stateless protocol, the switch can route any request to any server. This routing may be as simple as round-robbin, or it may be load-balanced based on measured response times to recent requests.
- Parallel servers have no need to communicate with one-another, and the only configuration a web server requires is knowing where to find the content it is serving.
- If a web-server crashes, the switch will stop sending it new requests. If the operations in the service protocol are idempotent, the client will time-out and retransmit the request, which will automatically be

routed to a different server.

Horizontally scalable systems do not have to be stateless. RESTful interfaces enable stateful interactions with all of the the same advantages. Even non-RESTful services can benefit from horizontal scaling. Consider the *shopping carts* supported by many product web sites. The front-end web-servers are indeed horizontally scaled and stateless. Shopping cart display and update operations are forwarded from the responding web-server to a back-end data-base server. The back-end data-base server may be highly stateful, and not at all horizontally scalable. But if 99.9% of all requests can be satisfied by the front line of web servers, it may be much easier to build a (perhaps vertically scaled) data-base server to handle the shopping cart requests.

Even (highly stateful) storage systems (including databases) can be designed to be horizontally scalable. Distributed Key/Value Stores often divide the key namespace into a independent subsets (usually called *shards*), each of which is assigned to a small group (e.g. 3) of servers. Assigning multiple servers to each shard enables us to maintain data availability even if some servers fail. Imposing a low cap on the number of servers involved in any particular transaction enables the protocols to scale to thousands of nodes with no degradataion in performance. When we need to add capacity, we do not add more servers for a particular shard; Rather we increase the number of shards into which the name space is divided, and assign the new shards to new servers. Similar techniques have been employed to improve the scalability of block storage, object storage, and even file storage services.

Cloud Model

The *cloud model* may be more a business model (renting services from an independent service provider, vs. buying equipment) than an architecture, but it is very well aligned with horizontally scalable architectures:

- All services are delivered via (well standardized) network protocols.
- The cloud model opaquely encapsulates the individual servers behind a single highly available IP address. This enables the network to distribute requests among the available servers.
- The resources presented by a cloud service are intended to be abstract/logical and thin-provisioned. Horizontally scaled architectures are well suited to implementing and delivering such services.
- The flexibility to continously add, remove, and rebalance resources in a horizontally scalable system, makes it possible for a cloud service provider to easily accommodate great fluctuations in demand.
- Public cloud services are (in most cases) unlikely to be co-located with the clients, and so the service protocols must be optimized for (throughput) efficiency over WAN-scale communications links.
- Horizontally scalable systems' ease of deployment/management yields economies of scale that make it possible for cloud service providers to offer their services at very competitive prices. Consider, for instance, the cost of maintaining redundant computing facilities in many different cities. This is normal operation for a large service provider, but would be prohibitively expensive for the typical company to do for itself.

But it is worth noting that the cloud model does not necessarily mean that all data access becomes remote. We always have a choice:

a. send the data to the computation (e.g. remote data access)
b. send the computation to the data (e.g. Map/Reduce)

Many cloud storage providers also also offer Virtual Machine hosting, and for applications that run in those remote virtual machines, all cloud data access will be local. For many applications, the amount of raw data processed is much greater than the amount of output that is produced. In such cases, it may be both faster and more economical to run the computation on a VM server that is co-located with the data it will process.