Welcome to the Machine

What every computer systems student should know about computers

Jeff Chase
Department of Computer Science
Duke University

January 11, 2013

Any systems course presumes a basic knowledge of computer architecture. This note gives an overview of the key concepts and vocabulary. For the supporting detail, the gold-standard source is *Computer Systems: A Programmer's Perspective*, popularly known as CS:APP, which gives an in-depth, nuts-and-bolts treatment of system programming with C/Unix on x86 computers.

This note reviews the basic concepts and terms for any conversation about operating systems and applications and how they run on a machine. The title should not be taken to suggest that this note contains everything you need to know: any serious systems student should take some time to get smart with sources like CS:APP. Developing a working understanding of the material in that book will place you with a fairly elite group of system-builders who truly understand how "real code runs on real metal".

But if the material here is familiar to you then you should be OK for an OS class. If it is *my* class, then these are the terms and meanings that I use to talk about machines. Be aware that others may use their terms slightly differently, and also that terms may have additional meanings in other conversations.

1 Processors

Any system that runs software has at least one processor (CPU) in it. A processor is a hardware element that executes software instructions in some machine language defined by its instruction set architecture (ISA). The CS:APP book focuses on the 32-bit Intel IA32 and 64-bit x86-64 ISAs, which are the basis for a large majority of servers and personal computers. In particular, it explains how C programs run on x86 machines. C is a good choice to study because there is a simple and direct mapping between C constructs and the ISA constructs. If you understand C, then you also understand most of what you need to know about how the machine stores and processes data. And if you understand how the machine works then it is easy to learn C. Ideally one can look at C code and "see" it from the point of view of the machine.

Processors are implemented on chips, which plug into sockets on hardware modules that implement the computer. Modern processor chips incorporate multiple cores (*multicore*). A core is a chunk of hardware circuitry that executes software instructions independently of other cores. The term *processor* may refer to either a chip or a core, so we use the terms chip (or socket) and core to avoid ambiguity. A chip with two cores is *dual-core*. A chip with four cores is *quad-core*. These configurations are common today.

The cores on a chip access main memory (RAM) external to the chip. The cores read and write memory through one or more levels of caches on the chip. The fastest cache level (L1) is private to each core (per-core). Depending on the chip, the lower cache levels (e.g., L3) may be shared by multiple cores. Each core also has a small array of named *registers* that store data in active use within the core, along with other state relating to the core's execution. The core ISA defines the specific registers and their names and uses.

Future chips will incorporate more cores and other elements as their densities continue to grow per Moore's Law. Computer architects will introduce new core groupings and cache structures as the chip technology evolves. Another trend is to combine different flavors of cores in the same chip, possibly including specialized elements optimized for specific tasks like graphics (GPUs) or cryptography: *heterogeneous multicore*.

2 Computers

A compute *node* consists of one or more processor chips sharing a memory. Each node runs its own copy of the operating system and may execute (and crash) independently of other nodes. Nodes are commonly linked by messaging networks. We call them "nodes" because they are independent but linked in a network structure.

It is common to manage a *cluster* of nodes as a unified computing facility in which the nodes are coordinated and/or interchangeable in some way. The nodes in a cluster are administered together and run common software. A supercomputer is a cluster linked by a special high-speed internal network, often called an *interconnect*.

Each node has one or more network interface controllers (NIC "nick" interface) on each network that the node attaches to. There may be multiple networks and multiple interfaces per network.

A node may have various other controllers that enable its software to control other *devices* in the computer and exchange data with them. The devices may include other I/O devices, such as disks or other storage, keyboards, screens, mice or other pointing devices, and so on. Devices called *sensors* report measurements of the external environment: cameras, microphones, GPS, etc. A *clock* monitors the passage of time. Other devices allow the software to monitor and control the basic health of the computer and respond to changes in its internal temperature, power supply, and so on.

The choice of devices and their interfaces and organization vary widely across different kinds of computers. A few points are common in general. A device can send an *interrupt* signal to a processor to request attention from the software. A processor can issue commands to each attached device. The command set is defined by the device type; the system software includes a *driver* module for each type of device. The driver contains code to interact with the device according to its specification. Devices that move a lot of data (e.g., NICs or disk controllers) generally use *Direct Memory Access (DMA)* to copy blocks of data in an out of the machine memory, leaving a processor free to do other work while the device moves the data. Command sets, formats, and other behaviors vary widely across devices.

3 Code

Processor cores execute machine instructions stored in memory. Software programs may be written in various languages and represented in various ways, but they are always translated into the underlying machine language to execute. The term "code" refers to a program or program fragment in any representation.

A software program that translates code from one representation to another is called a translator. In general, a *compiler* translates code before it begins to execute. An *interpreter* translates code as it executes. For example, a C compiler and related utility programs translate C source code files into *object files* containing compiled machine instructions. Another utility called a *linker* combines object files into an executable program file that can run under an operating system such as Unix.

In contrast, a Java compiler compiles Java classes into *class files* coded in a portable Java Virtual Machine (JVM) instruction set, which is then interpreted at runtime by a program called a JVM. As an optimization, the JVM interpreter may compile some fragments of the code into machine instructions as it interprets them. This technique is called just-in-time compilation (JIT).

4 Instructions and data

Machine instructions that reference the computer's memory address it as an array (sequence) of consecutively numbered byte locations. An *address* or *pointer* value is an offset into the byte array. The *address space* is the set of all addresses that executing code uses to access its memory.

It is useful to think of the machine instruction set as partitioned into three main groups of instructions.

Arithmetic instructions. These instructions perform basic arithmetic on values in the named registers. The various machine instructions interpret the values as instances of basic machine types, corresponding to the primitive C types `char, short, int, long, char*, float, and double`, and a few variations, e.g., signed and unsigned variants of the integer types. The machine ISA defines the sizes of these primitive types. [table]. The sizes are powers of two.

It is useful to understand the basic arithmetic operations and the corresponding C operators. They include bitwise logical operators (and, or, complement) that are important for systems code, e.g., code that interacts with devices. The shift-left and shift-right operators are useful to multiply or divide unsigned integers by powers of two. We may combine logical operators to select or clear specific bits in a value, e.g., for address calculations: this is called *masking*. C programmers should understand how the operators handle expressions that operate on mixes of signed and unsigned types—a common source of errors.

Load/store instructions. These instructions move data between memory and registers, i.e., to **load** a value from memory into a register or **store** a value from a register into memory. Any load or store instruction references memory at some given address (the *target*). Variants of the load and store instructions are suited to the different types, so that loads and stores transfer the correct number of bytes according to the type of the referenced value.

Branch instructions. These instructions determine control flow. A *branch* instruction may transfer control to the instruction at a given address (the branch target): after a branch the target instruction becomes the "next" instruction to execute. A *conditional branch* instruction determines whether or not to transfer control based on some arithmetic test of a value in a register (`<=> 0, != 0` etc.). If the checked condition is true then the branch is "taken", else the branch has no effect.

The target of a load, store, or branch instruction is typically given by an address value in a register. Some instructions specify the target as an offset (displacement) from a base address in a register. This is called indirect or base-plus-offset addressing. It enables software to reference selected fields of a block of data or code.

A *block* is a sequence of contiguous locations of memory starting at a base address. For example, a block might store a C structure or C++ object with named fields at specific offsets chosen by the compiler. The compiled code uses base-plus-offset addressing to reference the fields. Each named field has a different offset, which is known and coded into the instruction at compile time. The instruction has the intended effect if the correct base address is loaded into the base register when it executes.

5 Threads

As a core runs it fetches and executes instructions in some sequence. Specifically, the core executes instructions stored at successive addresses in memory unless it encounters a branch, which (if taken) transfers control to the instruction at the branch target. We may refer to a path of control flow through a program as a "code stream" or an "executing stream of instructions". A stream's path through a program is determined by the behavior of the instructions encountered and the values found in memory and in registers.

As a core executes a code stream, it uses a runtime *stack* to store data relating to procedure calls (method invocations, functions). The stack is a block of memory. Why do we need a stack? The code in any procedure may reference the procedure's local variables, parameters, and other local state. The system must maintain a separate instance of this local data for each instance of the procedure: for example, if the procedure is called recursively, then each invocation is a separate instance with a separate version of the procedure's local state. To address this need, each procedure instance has a temporary block of memory to store its local state while it executes. The block is called its *stack frame* or *activation record*. Each procedure *call* allocates a new stack frame and initializes it for the variables and parameters. Each procedure *return* deallocates the stack frame for the completed procedure instance, which is no longer needed. The call allocates the frame at the top of the runtime stack, "pushing" the frame on the stack. The return "pops" the frame from the stack.

We may refer to a stream and its stack together as a *thread*. We often speak loosely of a thread as an object that executes its stream of instructions. Actually it is a core that executes them, using the thread's stack, and following the thread's path through the program instructions stored in memory. Operating systems incorporate internal software primitives to manage threads, e.g., to start, stop, or suspend a thread, or switch a thread from one core to another. These primitives manipulate the state of the core to dispatch threads for execution, and tear them down.

Thus a thread is a *logical* execution stream of software, which the software itself may manipulate. This is a difficult concept. The key is to understand the processor context for a thread. The *context* is the set of values in the core registers, specifically in the subset of core registers that are involved in executing the thread on the core. The thread context includes at least the general registers used by the instructions encountered as the thread executes. The context also includes at least two designated registers:

- The *instruction pointer (IP)* or *program counter (PC)* register contains the address of the thread's "current" or "next" instruction to execute. As the core executes each instruction it advances the value in the PC to point to the next (successor) instruction automatically. A branch replaces the PC value with the branch target, which has the effect of transferring control to the target instruction.
- The *stack pointer (SP)* register contains a pointer to the top of the thread's runtime stack. Compiled procedure code references local variables for a procedure instance as fixed offsets from the SP. (It is a little different on x86 architectures, which have another related register called the *frame pointer (FP)*, but let us ignore that to keep it simple.) The procedure call/return code pushes and pops frames by adjusting the pointer value in the SP, subtracting or adding the stack frame size for that procedure.

System code may modify the context to manage use of the processor by concurrent and logically independent activities. The act of changing the context to switch the flow of execution is called a *context switch*. Context switch is implemented by a block of instructions that save selected register values of the active context into memory and/or load new values into core registers from memory.

For example, suppose a core is running a thread whose control flow enters a block of instructions to save the thread's context into memory and then transfer control somewhere else. If a second core then loads the saved context from memory, that core can continue executing the logical thread exactly where the first core left off. We say the first core *suspends* the thread and the second core *resumes* the thread. The code stream won't notice or care that it is running on a different core now. Once resumed, it executes the instruction at the address in the PC, which was the "next" instruction it was "supposed to" execute before it was suspended. The restarted instruction stream uses the values in the core's general registers, which are just as they were before. The stream uses the contents of the thread's stack, which is sitting in memory just as it was before, at the address in the SP register.

Of course this scenario presumes that the instructions, stack contents, and other data stored in memory have not changed in any way that causes the suspended thread to veer off in some unexpected direction when it resumes. Software must adhere to certain conventions and disciplines to ensure that threads execute correctly across suspend/resume and other concurrent execution. In particular, each thread must have its own private stack that is not modified by other threads. Also, each thread must execute on at most one core at any given time. There is no natural law that prevents a buggy system from loading the same saved context on two cores at the same time, but no good can come of it.

It is convenient to speak as if a core executes a single thread at a time. That was always true not so long ago.

Today some cores can execute two or more threads concurrently. They are called *multi-threaded cores.* A multi-threaded core must have a separate register context (a core slot) for each active thread. Supercomputer architects have experimented with novel many-threaded architectures that have dozens or hundreds of register sets per core. We will speak as if a core has only a single register set and executes at most one thread at a time, to simplify things without loss of generality. What we really mean is that the thread occupies a register set on the core and is active on the core.

6 Exceptions

When a stream/thread is executing on a core it has control of the core: the program and its data determine the path of the control flow. Processors include features to transfer control away from the executing stream when certain events occur. These events are called *exceptions.*

Exceptions are fundamental to system software because they enable an operating system to regain control of the core, i.e., divert the code stream so that it is running OS code. Once the OS software has control, it may perform system functions and/or transfer control to a different thread.

When a core detects that an exceptional event has occurred it is said to *raise* the exception to signal the event. The core transfers control to an event *handler* registered for the event type. The handler is a block of system code starting at some specified address. The operating system selects its handlers and registers their start addresses in a machine table when it starts up. The handlers run in a protected CPU mode (Section 9.1).

There are three kinds of exceptions: traps, faults, and interrupts.

- A *trap* results when the control flow encounters a special trap instruction, whose function is to raise the trap every time it executes. Software uses traps to transfer control deliberately to the operating system to request some service—a *system call.*
- A *fault* results from a condition detected in the machine while it executes some instruction. The fault may occur on some executions of that instruction, but not necessarily every time. A fault may indicate an error (e.g., divide by zero), or it may occur because some system software action is needed to emulate the instruction or otherwise enable it to execute.
- An *interrupt* results from some external condition, such as a clock tick or other signal from a device. Devices interrupt the processor when some event occurs on the device that requires software attention, such as an I/O request completion, an input event such as a network packet arrival, or some other change in status (e.g., power failure or device failure).

When an exception occurs the machine and/or handler save enough context to resume the previous code stream after the handler runs. The machine provides instructions for a handler to *return* from the exception to restore the saved context and resume what the stream was doing before. In some cases the handler modifies the saved context (e.g., the saved PC) before returning. For example, it may arrange to re-execute a faulting instruction, or advance to the next instruction (e.g., in the case of a trap), or abort the stream and discard its saved context. The latter response is common if the handler determines that the exception was triggered because of an error in the code stream.

Alternatively, the handler may modify the saved context to transfer control to another part of the program when it resumes. This technique is commonly used to notify the program itself of an exception that it caused. For example, a Unix program may register *signal handlers* for the system to invoke if certain conditions occur. Ultimately the signal handler might propagate the notification into the program using the exception mechanisms of programming languages like C++ or Java. Such higher-level "exception handlers" are not known to the system and are out of scope for this discussion.

7 Addressing

As noted earlier, the ISA supports instructions to operate on primitive machine types, corresponding to the basic C language types. Each primitive type has a size that is some power of two. The general registers are sized to store a value of any primitive type. Load and store instructions read and write values of various sizes, according to their types.

Code may view the memory as an array of any of these primitive types. More generally, code can interpret the bits in any piece of memory as values of any of these primitive types. A C program may interpret any address as a pointer to any base type, and operate on it accordingly. We might say that memory is "fungible". This is a common source of confusion for students programming "close to the metal" for the first time.

Modern programming languages carefully track the type of each location or value, and prevent a program from referencing any value in a manner inconsistent with its type. This property is called *type safety.* But the underlying machine language—and C—are more relaxed about how code interprets data in memory.

Changing the type of a value is called a *type cast.* Type casts are useful in system code to specify or change the interpretation of values in memory. A cast does not change the value itself, only how the code interprets the value. For example, when a device deposits data or a status report in memory, the system may overlay it with a `struct` type that defines names and types of the fields in the block, to interpret the information according to the device specifications. It does this by recasting the type of the block address as a pointer to the desired `struct`. Similarly, if a block of memory is repurposed from one use to another (e.g., by a dynamic heap manager) then the type of the structure previously stored there is forgotten, and the pointer is recast to a type for the block's new use. These operations in the C source code do not affect any values in memory: they are merely advisories to the compiler to influence how the code interprets these values.

7.1 Ways to get burned

Type casts are always unsafe. Sometimes they make sense, and sometimes they do not. Good programs use them sparingly. With great power comes great responsibility. Poor type enforcement and unsafe casting are a major source of security holes in real-world software.

Type casts sometimes happen "accidentally" through careless programming in C or other unsafe languages. For example, a common novice programming error is to save a pointer to a value that is stored on the stack (e.g., a local variable). Suppose some other code follows the pointer after the procedure returns and its stack frame is popped. Then the addressed location might now be part of some other frame that was pushed on the stack at some later time, and it may have an entirely different type and meaning. A similar problem occurs if the program frees a heap block while some other data structure still holds a pointer to a value in the heap block. These are examples of the menace of *dangling references*: old stale pointers referencing memory locations that have since been repurposed.

As another example, even today many programs copy string data from some external source (e.g., network input) into a block of memory allocated with some given size on the stack or heap. C supports variable-length strings; some versions of string copy (`strcpy` library routine) copy bytes from the string until a zero byte (null) is encountered, indicating the end of the string. If the string is longer than the allocated buffer, then the end of the string overwrites the memory locations past the end of the buffer, which the program may be using for some other purpose. This is called a *buffer overflow.* If the buffer is allocated as a local variable in a stack frame, then the tail of the string may overwrite data values of arbitrary types elsewhere on the stack. This can have catastrophic effects. Consider that when a procedure returns, it branches to a *return address* also stored in its stack frame. If the return address is overwritten by a long string, then the program branches to an arbitrary location determined by the contents of the string. This simple error creates an opening for a deliberate *stack smash* attack by a malicious client. These attacks have caused untold pain and suffering in the real world for decades. Be smart and use `strncpy`.

7.2 Alignment

In general, a load or store on a value of a given type should be aligned for the type's size: the target address should be a multiple of the size of the type. Equivalently, the instruction references its target as if the program views memory as an array of elements of that type, and is referencing a cell in that array. Some processors raise an exception on an unaligned load or store. Other processors can execute unaligned loads and stores correctly, but at a slower rate.

Compilers and some other system software must be aware of the need to align data in memory. Blocks of data in memory may combine elements of different types in various orders (e.g., C structs, parameters for a procedure, etc.). The compiler or interpreter translates between symbols (e.g., variable names) used in the program and the underlying addresses of the named data elements. The compiler or interpreter may insert unused bytes to pad the data as needed to respect alignment constraints for all elements. Similarly, heap managers must allocate requested dynamic memory blocks at addresses aligned for the largest machine type that might be stored in those heap blocks.

8 Block I/O

It is common for disk I/O or network I/O devices to use DMA to transfer data to/from memory in fixed-size blocks. We can view memory as an array of blocks of any fixed size. Similarly, we may also view the contents of a disk or other storage device as an array of fixed-size blocks.

Our world is simpler when these blocks have power-of-two sizes and are aligned in memory and on disk. They generally are, so we presume it without loss of generality. For example, file systems and databases are organized on disk within fixed-size blocks at aligned addresses on disk. The file or database system software orchestrates the movement of these blocks between storage devices and aligned locations in memory.

Note. We refer to storage as "disk", but it may be implemented using any technology. We assume only that a storage device is accessed through a DMA device interface, that the device commands may address the storage device as a sequence of blocks, and that the storage is non-volatile, i.e., any stored values persist across system restarts, power failures, and the like. Any such storage device may host a database or file system.

When we view memory and storage as arrays of fixed-size aligned blocks, addresses and offsets have a natural structure. The high-order bits of any byte address encode a *block number*—an index into the block array, uniquely addressing a specific block. The low-order bits encode a *byte offset* of a byte within the block. Shift and mask operations are useful for doing fast arithmetic on addresses within such a block space.

For example, consider an array of 4KB blocks within a 4GB space, whether it is on a storage device or in memory. The entire space is byte-addressable with 32-bit addresses. There are 1M blocks in the space. A block number is represented as a 20-bit quantity. A 32-bit address of a byte has a 20-bit block number in the high 20 bits. The lower 12 bits of an address contain a byte offset into the named 4KB block.

Data structures in storage blocks often contain embedded references to other blocks on the device. In this way it is possible to build the equivalent of dynamic pointer structures on disk, and access them in memory from software by following the on-disk pointers, initiating I/O to move blocks to and from memory as needed. This is useful because storage structures are inherently dynamic: like a heap manager, a file system must handle streams of requests to create/delete files of various sizes. And files grow.

However, the software must manage these dynamic storage structures carefully: if the system crashes while an operation on a disk-based structure is in progress, then the structure may become corrupted. The issue of how to manage recoverable structures on persistent storage is an important topic in operating systems and databases.

9 Virtual address spaces

I said earlier that the "address space" is the set of all addresses that an executing program uses to access its memory. Of course a given address used by a program does not necessarily correspond to fixed location in the machine's memory.

Trusted software deep in the operating system does use machine addresses. But most software runs within some *virtual address space* that presents a custom window into the machine memory for use by a specific software context. There may be many virtual address spaces active in the system at any given time. Threads execute within virtual address spaces. The system can switch between virtual address spaces just as it switches between threads. Multiple threads may execute within the same virtual address space.

The memory referenced through a virtual address space is "virtual" because a thread executing within the space does not know where its data is actually stored in machine memory, or even if the data is actually resident in the machine memory at all. Virtual address spaces provide isolation: a thread cannot access any area of machine memory unless the system attaches that memory to its virtual address space.

Virtual addressing works as follows.

- The machine memory is viewed as an array of fixed-size n-byte memory blocks called *page frames*, or just *frames*. Each index into this array is called a *page frame number (PFN).*
- Each virtual address space is viewed as an array of n-byte *virtual pages*, or just *pages*. Each index into this array is called a *virtual page number (VPN).*
- The system assigns pages to frames and maintains a mapping from pages to frames. A "mapping" is a function: each page maps to at most one frame, and each frame holds at most one page. Frames act as interchangeable slots to store pages. The system stores the mapping in an internal table called a *translation table* or *page table.*
- Virtual addresses are translated to machine addresses during execution, by replacing the VPN in the higher bits with the corresponding PFN, which is given by some translation in the table. The machine provides a cache of translations (a TLB) to speed it up. If the needed translation is cached in the TLB then the machine does all of the work. The machine support for virtual addressing is referred to generically as its *memory management unit* (MMU).
- If the translation table does not have a valid translation enabled, then the MMU raises a fault—a *page fault.* The system handler for a page fault may consider the reference an error, or it may assist in completing the reference.

Note that frames are raw memory, while pages have values. A virtual page is a block of values that some code expects to appear at a particular location (block) within its virtual address space, with the correct values. If a thread references a virtual address within the page, then the reference can complete only if the page's value (its contents) resides within some frame, and a valid $VPN \to PFN$ translation is present in the translation table.

If a translation is not present, then the page fault handler must decide how to handle the situation. The handler may resolve the fault and restart the faulting instruction. To resolve the fault it may install or remove translations to change the mapping of pages to frames, store data into a frame to initialize the page contents (e.g., to zero it), or initiate I/O to read a page from storage into a frame.

Different machines implement the translation table in different ways, and the hardware details vary widely. Conceptually, the system assigns a unique number to identify each virtual address space, called its *address space identifier* or ASID. Or (equivalently) the base address of the page table for the virtual address space serves as its ASID. The translation table implements a function with a partial mapping $(ASID, VPN) \to PFN$. We can view the subset of translations with a given $ASID$ as a logically separate translation table for the named virtual address space. On x86 systems the operating system does in fact maintain a separate page table structure for each virtual each address space.

Trusted system software controls the translations that are in effect at any given time, e.g., by storing the ASID in a special register. To understand how it works, we first need a better understanding of what it means for system software to be "trusted".

9.1 Protected mode

Processor architectures support one or more modes of protected execution. Certain instructions and operations are permitted only if the core is executing in the proper mode (privilege level). For example, control over devices and translation tables are reserved to a protected mode. Certain registers are protected: software can modify them only if the core is running in protected mode. For example, the ASID for the current context is stored in a protected register. In fact, the current mode of the processor is also stored in a protected register.

For example, Unix presumes that the machine has at least a one-bit protected mode with two states called *kernel mode* and *user mode*. The system software that runs in kernel mode is called the *kernel*. The kernel is similar in many respects to any other software program: it is written in C or perhaps some other high-level language, and consists of modules or components with procedural interfaces and internal data structures. But it can control all functions of the machine because it runs with the CPU in protected kernel mode.

When a CPU core runs an untrusted user program it runs in user mode. The core is loaded with an ASID for a virtual address space. When a core is running in user mode with a virtual address space we say that it is running in a *user context*. The virtual address maps limit the visibility of machine memory: the executing code stream can access only its own program instructions and any memory that the kernel allocates to the program instance for its own use. In general, the virtual address maps prevent the core from accessing kernel code or kernel data structures when it is running in user mode with a virtual address space. Kernel code executes within a logically separate virtual address space that cannot be accessed in user mode. (Details vary.)

We commonly use the term "user space" or "user level" to refer to any code or data accessed through a virtual address space in user mode, or more generally for any execution in user mode. We commonly use the term "kernel space" to refer to any code or data accessed only through the kernel address space in kernel mode, or more generally for any execution in kernel mode.

We may presume that all software runs the machine in either kernel mode or user mode. Note, however, that some systems use additional protected modes. In particular, modern x86 hardware support multiple *virtual machines* (VMs) running on the same physical computer. Each VM behaves as described here: it runs an operating system instance with a kernel mode and a user mode. System virtualization software called a *hypervisor* or *virtual machine monitor* (*VMM*) runs outside of the virtual machines: it runs the processor in another protected mode that enables it to manage how those VMs use the physical hardware. Section 9.5 discusses hypervisors and virtual machine management. Until then we focus on the machine model for Unix and other operating systems, in which the machine runs in either kernel mode or user mode.

9.2 Crossing the protection boundary

All of this brings us to a crucial point for understanding operating systems. Once the core is running in a user context, it is limited in what it can do. It cannot execute protected instructions or access protected registers. It cannot access devices without kernel intervention. It cannot break out of its virtual address space: it can access a given memory page frame only if the kernel permits it, i.e., by installing a translation to that frame in the page table for that space. Software in user mode cannot change its own translations: the ASID for the current page table is stored in a protected register, and the page table itself is a kernel data structure and so is not accessible through the address space.

The core executes safely and happily in user mode unless and until a machine exception occurs: a trap, fault, or interrupt (Section 6). A machine exception returns control to the kernel. Specifically, if the core raises an exception the machine transfers control to a kernel handler registered for exceptions of that kind. The handler runs in kernel mode.

The handler is the entry point for the kernel to decide how to handle the exception, as summarized in Section 6. On entry to the handler the machine provides a reference to a status block describing the exception and the register values at the time of the exception. The handler can determine what caused the exception and the context in which the exception occurred, including the space and thread. The handler takes appropriate action:

- If the program took an illegal action to cause the exception, then the kernel may kill the program and tear down its address space, or it may notify the program by resuming execution in user mode at the address of a handler routine registered by the program (e.g., a Unix signal handler).
- If the exception is a page fault resulting from a legal virtual memory reference, then the kernel may modify the page table to install a new translation, and then resume execution in user mode at the address of the faulting instruction, allowing the reference to complete. The kernel may choose to suspend the thread and resume it later.
- If the exception is a trap requesting a system call, then the registers contain a syscall ID and the arguments to the system call. The syscall ID is an integer indicating which system call is requested. The kernel validates the request and arguments, and attempts to provide the requested service by calling a corresponding API routine in kernel mode.
- If the exception is an interrupt, then the interrupt handler attends to housekeeping tasks associated with the interrupt. For example, it may record completion of an I/O request to a device, or receive notification of a new input such as an arriving network packet, screen tap, keystroke, or mouse click. The kernel also receives *timer interrupts* to inform it of the passage of time. After handling the interrupt, the kernel resumes the interrupted context, or switches to some other context if it chooses.

In all cases the kernel may choose to suspend the executing thread and resume it later. In particular, many system calls request the kernel to suspend the calling thread until some event occurs. For example, the system call may request a synchronous I/O operation: the kernel initiates the I/O and suspends the calling thread until the I/O completes. The kernel may suspend a thread for its own reasons. In particular, the kernel may choose to reallocate the core for use by another thread.

These crucial mechanisms for safe transitions in and out of protected mode are the bedrock of operating systems, because they enable the operating system—trusted system code—to protect itself from untrusted applications, protect/isolate user programs from one another, and mediate access to the machine resources. In brief, code running in protected mode can drop out of it at will, but the core enters protected mode only on an exception, which always transfers control to a trusted system handler. Malicious software cannot take control of the system unless it can circumvent this protection mechanism. If it succeeds then all bets are off.

9.3 An unusual metaphor

We can envision these kernel/user transitions as a game of tennis or ping-pong among multi-armed juggling robots. Use your imagination and bear with me.

The machine is a tennis court with a net: the net is the kernel/user boundary. The kernel robot stands on one side of the boundary. The robot players on the other side are the user contexts—executing instances of user programs. An executing program instance is called a *process* in Unix and other systems.

Any of the players may run multiple threads. Each user process robot has an arm holding a paddle for each of its threads. The kernel has an arm and paddle for every thread. Each thread corresponds to a ball and a pair of paddles: the kernel holds one of the paddles, and the process that owns the thread holds the other paddle. A robot may hit a ball only with a paddle that is associated with that ball. A paddle corresponds to a stack for the ball to run on while it is in that player's control.

The kernel holds all of the power in this game:

- The kernel always serves. Every ball starts on the kernel side and is served by the kernel to a user process. The process juggles the ball for some period of time and then hits it back to the kernel. The hit is an exception. The kernel receives each ball as it crosses back over the net, juggles it for a time, and eventually either hits it back to its user context or drops it on the ground.
- The kernel always controls the ball. If a process is juggling a ball, the kernel can blow a whistle at any time and force it to hit that ball back over the net. (The whistle corresponds to a timer interrupt on the core, initiating a context switch.)
- The kernel can also hide a ball in its pocket and launch it out again later at will. The game can only have N balls in the air at a time, where N corresponds to the number of core slots on the machine. The kernel hides the balls as needed to preserve this property. (Hiding a ball corresponds to suspending a thread until the kernel chooses to run it again.)
- Also, the kernel can create and destroy opponents (processes) with its mind. It can will a new process into existence on the other side of the net just by imagining it. It snaps its fingers and a ball appears to serve to the new opponent, along with an arm and paddle to make the serve. If a process displeases the kernel it simply vanishes from the game and all of its threads fall on the ground.
- If the kernel makes a mistake it may choose to panic and restart the system: the balls all fall on the ground, the court is wiped clean, and the game begins anew.

A few important points emerge from this picture. Threads move back and forth between the kernel and a user process context. A process can juggle the ball as long as it wants, but it ultimately it can do nothing else but hit the ball back over the net to the kernel. The processes cannot hit balls to or at one another. Only the kernel decides when the game begins or ends, or who plays, or how many balls are on the court, or when or whether to hit them back. A process may ask to be dismissed (exit), or ask the kernel to create and destroy other processes or threads, but ultimately the kernel decides the course of the game. And, of course, being the kernel means never having to say you're sorry if you drop the ball, hide the ball, or vaporize your opponent.

The kernel regains control of the core on exceptions: traps, faults, and interrupts. For example, suppose code in user space attempts to access a protected instruction or register, or references a virtual address for which its page table gives no valid translation. The core raises a fault and transfers control to a system handler accordingly: the ball comes back over the net to the kernel side. The system handler may treat the exception as an error, or it may provide some operating system service or emulation before hitting the ball back to user space.

9.4 Handler stacks

Exception handlers may call procedures, and so a handler needs a stack to run on. Traps and faults occur synchronously with thread execution, i.e., as a result of the thread executing a specific instruction. This means that in general each thread needs its own stack to run handlers for any exceptions raised during its execution. For example, two concurrent threads running on different cores can trap for a system call at the same time; in this case, their handlers execute independently and concurrently, and each must have its own stack. Moreover, exception handlers are system code, and their stacks must be protected for use by the system only.

For example, in a Unix system, each thread or process has a stack in kernel space for handling traps and faults. The kernel stack is separate from the thread's stack in user space, and is separate from the kernel stacks of all other threads. In general, the stack pointer to use for a trap or fault handler is a separate and protected part of the register context.

Interrupts are independent of any thread, and so their handlers generally execute on a system-wide interrupt stack. For example, there may be a single interrupt stack for each core that can run interrupt handlers. Interrupts may be nested: one interrupt may occur while another is still being handled. Care is taken to size the interrupt stack with sufficient space for the maximum call depth of the nested handlers.

9.5 Advanced protection features

Since 2008, most x86-64 processors have additional protected modes to implement virtual machines efficiently. On these systems it is possible to host multiple isolated virtual machines, which are often called *guests* or *tenants*. For our purposes, a *virtual machine* is a context that supports all of the machine features needed to run a protected operating system with accompanying application software.

Modern virtual machines used in virtual compute clouds run “native” on the hardware, without software emulation. The term “virtual machine” is also used to refer to a JVM or other interpreter that emulates some abstract machine model in software. These variants of “virtual machine” look similar from the viewpoint of software running within the virtual machine, but they are implemented quite differently.

Processor architects developed new protection modes and related MMU extensions to support native virtual machines during the 2000s. Each virtual machine may run its own *guest* operating system instance, including a kernel that is trusted only within that virtual machine. Trusted software called a *hypervisor* orchestrates the virtual machines and mediates their access to the hardware. Everything said in this note about “machines” applies to native virtual machines as well.

Most machines today also have a tamper-evident secure co-processor such as a *trusted platform module (TPM)* supporting various cryptographic functions and other functions to protect or verify system integrity. These are topics for an advanced class.