I An Introduction to Memory Forensics

1 Systems Overview

This chapter provides a general overview of the hardware components and operating system structures that affect memory analysis. Although subsequent chapters discuss implementation details associated with particular operating systems, this chapter provides useful background information for those who are new to the field or might need a quick refresher. The chapter starts by highlighting important aspects of the hardware architecture and concludes by providing an overview of common operating system primitives. The concepts and terminology discussed in this chapter are referred to frequently throughout the remainder of the book.

Digital Environment

This book focuses on investigating events that occur in a digital environment. Within the context of a digital environment, the underlying hardware ultimately dictates the constraints of what a particular system can do. In many ways, this is analogous to how the laws of physics constrain the physical environment. For example, physical crime scene investigators who understand the laws of physics concerning liquids can leverage bloodstains or splatter patterns to support or refute claims about a particular crime. By applying knowledge about the physical world, investigators gain insight into how or why a particular artifact is relevant to an investigation. Similarly, in the digital environment, the underlying hardware specifies the instructions that can be executed and the resources that can be accessed. Investigators who can identify the unique hardware components of a system and the impact those components can have on analysis are in the best position to conduct an effective investigation.

On most platforms, the hardware is accessed through a layer of software called an *operating system*, which controls processing, manages resources, and facilitates communication with external devices. Operating systems must deal with the low-level details of the particular processor, devices, and memory hardware installed in a given system.

Typically, operating systems also implement a set of high-level services and interfaces that define how the hardware can be accessed by the user's programs.

During an investigation, you look for artifacts that suspected software or users might have introduced into the digital environment and try to determine how the digital environment changed in response to those artifacts. A digital investigator's familiarity with a system's hardware and operating system provide a valuable frame of reference during analysis and event reconstruction.

PC Architecture

This section provides a general overview of the hardware basics that digital investigators who are interested in memory forensics should be familiar with. In particular, the discussion focuses on the general hardware architecture of a personal computer (PC). We primarily use the nomenclature associated with Intel-based systems. It is important to note that the terminology has changed over time, and implementation details are constantly evolving to improve cost and performance. Although the specific technologies might change, the primary functions these components perform remain the same.

NOTE

We generically refer to a PC as a computer with an Intel or compatible processor that can run Windows, Linux, or Mac OS X.

Physical Organization

A PC is composed of printed circuit boards that interconnect various components and provide connectors for peripheral devices. The main board within this type of system, the *motherboard*, provides the connections that enable the components of the system to communicate. These communication channels are typically referred to as *computer busses*. This section highlights the components and busses that an investigator should be familiar with. Figure 1-1 illustrates how the different components discussed in this section are typically organized.

CPU and MMU

The two most important components on the motherboard are the processor, which executes programs, and the main memory, which temporarily stores the executed programs and their associated data. The processor is commonly referred to as the *central processing unit* (*CPU*). The CPU accesses main memory to obtain its instructions and then executes those instructions to process the data.

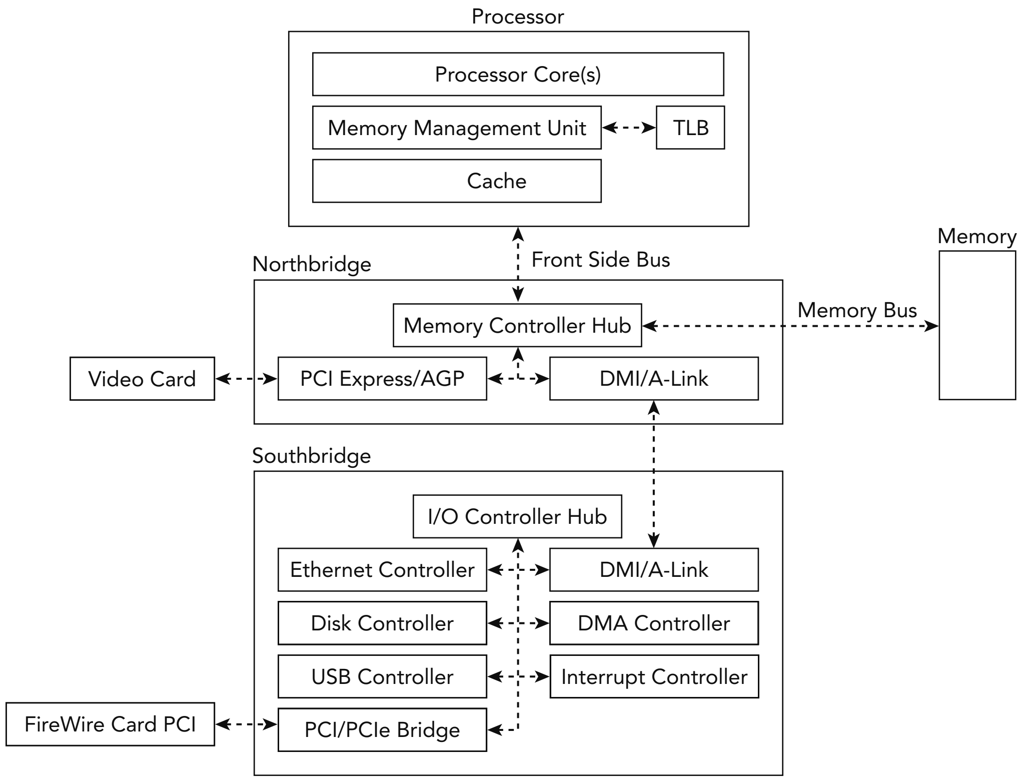

Figure 1-1: Physical organization of a modern system

Reading from main memory is often dramatically slower than reading from the CPU's own memory. As a result, modern systems leverage multiple layers of fast memory, called *caches*, to help offset this disparity. Each level of cache (L1, L2, and so on) is relatively slower and larger than its predecessor. In most systems, these caches are built into the processor and each of its cores. If data is not found within a given cache, the data must be fetched from the next level cache or main memory.

The CPU relies on its *memory management unit* (*MMU*) to help find where the data is stored. The MMU is the hardware unit that translates the address that the processor requests to its corresponding address in main memory. As we describe later in this chapter, the data structures for managing address translation are also stored in main memory. Because a given translation can require multiple memory read operations, the processor uses a special cache, known as the *translation lookaside buffer* (*TLB*), for the MMU translation table. Prior to each memory access, the TLB is consulted before asking the MMU to perform a costly address translation operation.

Chapter 4 discusses more about how these caches and the TLB can affect forensic acquisition of memory evidence.

North and Southbridge

The CPU relies on the *memory controller* to manage communication with main memory. The memory controller is responsible for mediating potentially concurrent requests for system memory from the processor(s) and devices. The memory controller can be implemented on a separate chip or integrated within the processor itself. On older PCs, the CPU connected to the *northbridge* (memory controller hub) using the *front-side-bus* and the northbridge connected to main memory via the *memory bus*. Devices (for example, network cards and disk controllers) were connected via another chip, called the *southbridge* or input/output controller hub, which had a single shared connection to the northbridge for access to memory and the CPU.

To improve performance and reduce the costs of newer systems, most capabilities associated with the memory controller hub are now integrated into the processor. The remaining chipset functionality, previously implemented in the southbridge, are concentrated on a chip known as the platform controller hub.

Direct Memory Access

To improve overall performance, most modern systems provide I/O devices the capability to directly transfer data stored in system memory without processor intervention. This capability is called *direct memory access* (*DMA*). Before DMA was introduced, the CPU would be fully consumed during I/O transfers and often acted as an intermediary. In modern architectures, the CPU can initiate a data transfer and allow a DMA controller to manage the data transfer, or an I/O device can initiate a transfer independent of the CPU.

Besides its obvious impact on system performance, DMA also has important ramifications for memory forensics. It provides a mechanism to directly access the contents of physical memory from a peripheral device without involving the untrusted software running on the machine. For example, the PCI bus supports devices that act as *bus masters*, which means they can request control of the bus to initiate transactions. As a result, a PCI device with bus master functionality and DMA support can access the system's memory without involving the CPU.

Another example is the IEEE 1394 interface, commonly referred to as *Firewire*. The IEEE 1394 host controller chip provides a peer-to-peer serial expansion bus intended for connecting high-speed peripheral devices to a PC. Although the IEEE 1394 interface is typically natively found only on higher-end systems, you can add the interface to both desktops and laptops using expansion cards.

Volatile Memory (RAM)

The main memory of a PC is implemented with *random access memory* (*RAM*), which stores the code and data that the processor actively accesses and stores. In contrast with sequential access storage typically associated with disks, random access refers to the characteristic of having a constant access time regardless of where the data is stored on the media. The main memory in most PCs is dynamic RAM (DRAM). It is dynamic because it leverages the difference between a charged and discharged state of a capacitor to store a bit of data. For the capacitor to maintain this state, it must be periodically refreshed—a task that the memory controller typically performs.

RAM is considered *volatile memory* because it requires power for the data to remain accessible. Thus, except in the case of cold boot attacks (`https://citp.princeton.edu/research/memory`), after a PC is powered down, the volatile memory is lost. This is the main reason why the "pull the plug" incident response tactic is not recommended if you plan to preserve evidence regarding the system's current state.

CPU Architectures

As previously mentioned, the CPU is one of the most important components of a computer system. To effectively extract structure from physical memory and understand how malicious code can compromise system security, you should have a firm understanding of the programming model that the CPU provides for accessing memory. Although the previous section focused on the physical organization of the hardware, this section focuses on the logical organization exposed to the operating system. This section begins by discussing some general topics that pertain to CPU architectures and then highlights the features relevant to memory analysis. In particular, this section focuses on the 32-bit (IA-32) and 64-bit (Intel 64) organization, as specified in the *Intel 64 and IA-32 Architecture Software Developer's Manual* (`http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf`).

Address Spaces

For the CPU to execute instructions and access data stored in main memory, it must specify a unique address for that data. The processors discussed in this book leverage byte addressing, and memory is accessed as a sequence of bytes. The *address space* refers to a range of valid addresses used to identify the data stored within a finite allocation of memory. In particular, this book focuses on systems that define a byte as an 8-bit quantity. This addressing scheme generally starts with byte 0 and ends at the offset of the final byte of memory in the allocation. The single continuous address space that is exposed to a running program is referred to as a *linear address space*. Based on the memory models discussed in the book and their use of paging, we use the terms *linear addresses* and *virtual addresses* interchangeably. We use the term *physical address space* to refer to the addresses that the processor requests for accessing physical memory. These addresses are obtained by translating the linear addresses to physical ones, using one or more page tables (discussed in more detail soon). The following sections discuss how memory address spaces are implemented in different processor architectures.

NOTE

When dealing with raw, padded memory dumps (see Chapter 4), a physical address is essentially an offset into the memory dump file.

Intel IA-32 Architecture

The IA-32 architecture commonly refers to the family of x86 architectures that support 32-bit computation. In particular, it specifies the instruction set and programming environment for Intel's 32-bit processors. The IA-32 is a little endian machine that uses byte addressing. Software running on an IA-32 processor can have a linear address space and a physical address space up to 4GB. As you will see later, you can expand the size of physical memory to 64GB using the IA-32 *Physical Address Extension* (*PAE*) feature. This section and the remainder of the book focuses on protected-mode operation of the IA-32 architecture, which is the operational mode that provides support for features such as virtual memory, paging, privilege levels, and segmentation. This is the primary state of the processor and also the mode in which most modern operating systems execute.

Registers

The IA-32 architecture defines a small amount of extremely fast memory, called *registers*, which the CPU uses for temporary storage during processing. Each processor core contains eight 32-bit general-purpose registers for performing logical and arithmetic operations, as well as several other registers that control the processor's behavior. This section highlights a few of the control registers relevant for memory analysis.

The EIP register, also referred to as the program counter, contains the linear address of the next instruction that executes. The IA-32 architecture also has five control registers that specify configuration of the processor and the characteristics of the executing task. CR0 contains flags that control the operating mode of the processor, including a flag that enables paging. CR1 is reserved and should not be accessed. CR2 contains the linear address that caused a page fault. CR3 contains the physical address of the initial structure used for address translation. It is updated during context switches when a new task is scheduled. CR4 is used to enable architectural extensions, including PAE.

Segmentation

IA-32 processors implement two memory management mechanisms: *segmentation* and *paging*. Segmentation divides the 32-bit linear address space into multiple variable-length segments. All IA-32 memory references are addressed using a 16-bit segment selector, which identifies a particular segment descriptor, and a 32-bit offset into the specified segment. A segment descriptor is a memory-resident data structure that defines the location, size, type, and permissions for a given segment. Each processor core contains two special registers, GDTR and LDTR, which point to tables of segment descriptors, called the *Global Descriptor Table* (GDT) and the *Local Descriptor Table*, respectively. The segmentation registers CS (for code), SS (for stack), and DS, ES, FS, and GS (each for data) should always contain valid segment selectors.

While segmentation is mandatory, the operating systems discussed in this book hide segmented addressing by defining a set of overlapping segments with base address zero, thereby creating the appearance of a single continuous "flat" linear address space. However, segmentation protections are still enforced for each segment, and separate segment descriptors must be used for code and data references.

NOTE

Because most operating systems do not take advantage of more sophisticated IA-32 segmentation models, segmented addressing is disabled in 64-bit mode. In particular, segment base addresses are implicitly zero. Note that segmentation protections are still enforced in 64-bit mode.

Paging

Paging provides the ability to virtualize the linear address space. It creates an execution environment in which a large linear address space is simulated with a modest amount of physical memory and disk storage. Each 32-bit linear address space is broken up into fixed-length sections, called pages, which can be mapped into physical memory in an arbitrary order. When a program attempts to access a linear address, this mapping uses memory-resident *page directories* and *page tables* to translate the linear address into a physical address. In the typical scenario of a 4KB page, as shown in Figure 1-2, the 32-bit virtual address is broken into three sections, each of which is used as an index in the paging structure hierarchy or the associated physical page.

The IA-32 architecture also supports pages of size 4MB, whose translation requires only a page directory. By using different paging structures for different processes, an operating system can provide each process the appearance of a single-programmed environment through a virtualized linear address space. Figure 1-3 shows a more detailed breakdown of the bits that translate a virtual address into an offset in physical memory.

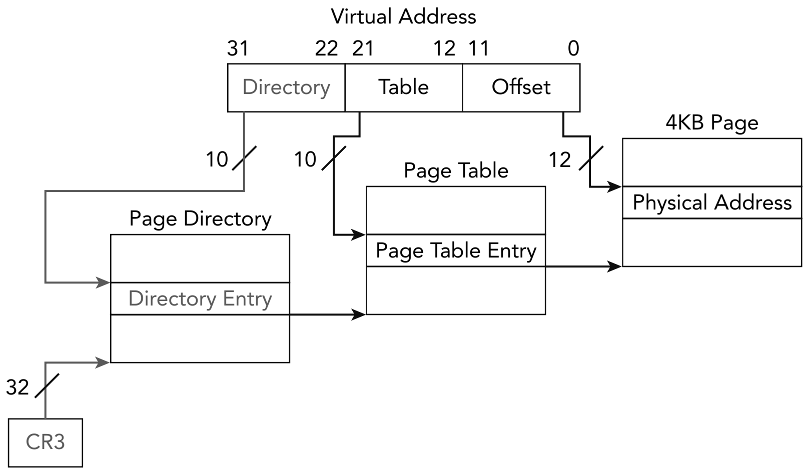

Figure 1-2: Address translation to a 4KB page using 32-bit paging

```
3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
```

PDE	CR3[31:12]		VA[31:22]	0	0
4MB Page	PDE[31:22]	VA[21:0]			
PTE	PDE[31:12]		VA[21:12]	0	0
PA	PTE[31:12]		VA[11:0]		

Figure 1-3: Formats for paging structure addresses used in 32-bit paging

To compute the page directory entry (PDE) address, you combine bits 31:12 from the CR3 register with bits 31:22 from the virtual address. You then locate the page table entry (PTE) by combining bits 31:12 from the PDE with bits 21:12 of the virtual address. Finally, you can obtain the physical address (PA) by combining bits 31:12 of the PTE with bits 11:0 of the virtual address. You'll see these calculations applied in the next section as you walk through translating an address manually.

Address Translation

To fully support a CPU architecture that offers virtual memory, memory forensics software such as Volatility must emulate the virtual address space and transparently handle virtual-to-physical-address translation. Walking through the address translation steps manually helps solidify your understanding of how these tools work and provides the background to troubleshoot unexpected behavior.

NOTE

The Python classes in Volatility that handle address translation expose a method called `vtop` (virtual to physical). Callers pass the function a virtual address and it returns the physical offset, which it computes using the steps described in this section. Similarly, if you're working with Microsoft's debugger (WinDbg), you can use the `!vtop` command.

For the sake of this exercise, we assume you are analyzing one of the memory samples, ENG-USTXHOU-148, included in Jack Crook's November 2012 forensics challenge (see `http://blog.handlerdiaries.com/?p=14`). During your analysis, you found a reference to a virtual address, `0x10016270`, within the virtual address space of the `svchost.exe` process with PID 1024. The page directory base (CR3) for PID 1024 is `0x7401000`. You want to find the corresponding physical address to see what other data might be in close spatial proximity.

Your first step is to convert the virtual address, `0x10016270`, from hexadecimal to binary format because you will be working with ranges of address bits:

```
0001 0000 0000 0001 0110 0010 0111 0000
```

Next, you decompose the address into the relevant offsets that are used during the translation process. This data is shown in Table 1-1.

Table 1-1: A Breakdown of the Bits for Virtual Address Translation

Paging Structure	VA Bits	Binary	Hex
Page directory index	Bits 31:22	0001000000	0x40
Page table index	Bits 21:12	0000010110	0x16
Address offset	Bits 11:0	001001110000	0x270

As seen in Figure 1-2 and Figure 1-3, you can calculate the physical address of the PDE by multiplying the page directory index by the size of the entry (4 bytes) and then adding the page directory base, `0x7401000`. The 10 bits from the virtual address can index 1024 (2^{10}) entries in the page directory.

PDE address = 0x40 * 4 + 0x7401000 = 0x7401100

Next, you must read the value from physical memory stored at the PDE address. Make sure to account for the fact that the value is stored in a little endian format. At this point, you know the value of the PDE is `0x17bf9067`. Based on Figure 1-3, you know that bits 31:12 of the PDE provide the physical address for the base of the page table. Bits 21:12 of the virtual address provide the page table index because the page table is composed of 1024 (2^{10}) entries. You can calculate the physical address of the PTE by multiplying the size of the entry (4 bytes) by the page table index and then adding that value to the page table base.

PTE address = 0x16 * 4 + 0x17bf9000 = 0x17bf9058

The value of the PTE stored at that address is `0x170b6067`. Based on Figure 1-3, you know that bits 31:12 of the physical address are from the PTE and bits 11:0 are from the virtual address. Thus, the final converted physical address is this:

Physical address = 0x170b6000 + 0x270 = 0x170b6270

After completing the translation, you found that the virtual address `0x10016270` translates to the physical address `0x170b6270`. Figure 1-4 provides a graphical illustration of the steps that were involved. You can find that byte offset in the memory sample and look for any related artifacts that might be in close proximity. This is the same process that the Volatility `IA32PagedMemory` address space performs every time a virtual address is accessed. In the following text, you see how this process can be extended to support larger virtual address spaces.

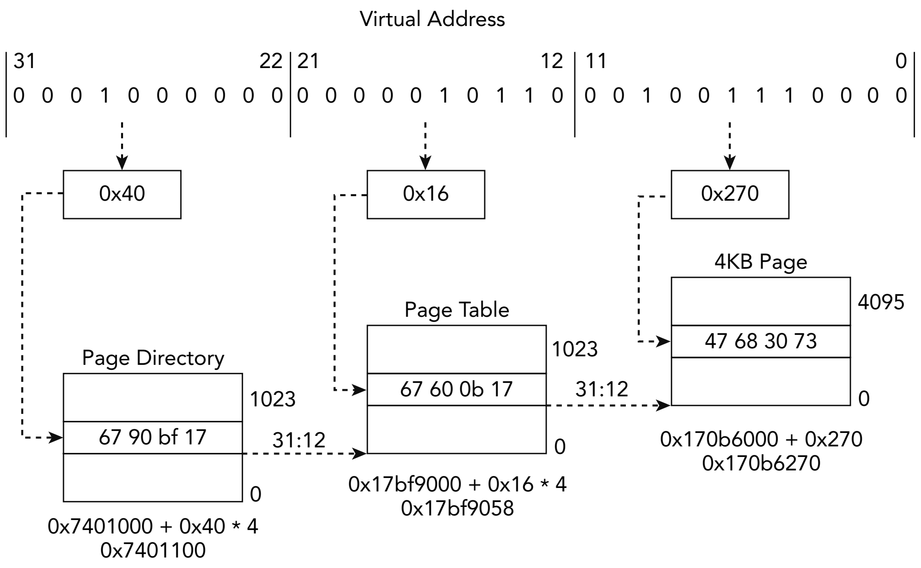

Figure 1-4: Example address translation to a 4KB page using 32-bit paging

NOTE

It is also important to highlight a couple of the bit flags stored in paging structure entries that directly affect translation for all three paging modes discussed in the book. The address translation process will terminate if a paging structure entry has bit 0 (the present flag) set to 0, which signifies "not present." Thus, it generates a page fault exception. If you are processing an intermediary paging structure, meaning more than 12 bits remain in the linear address, bit 7 of the current paging structure entry is used as the page size (PS) flag. When the bit is set, it designates that the remaining bits map to a page of memory as opposed to another paging structure.

Physical Address Extension

The IA-32 architecture's paging mechanism also supports PAE. This extension allows the processor to support physical address spaces greater than 4GB. Although programs still possess linear address spaces of up to 4GB, the memory management unit maps those addresses into the expanded 64GB physical address space. On systems with PAE enabled, the linear address is divided into four indexes:

- Page directory pointer table (PDPT)
- Page directory (PD)
- Page table (PT)
- Page offset

Figure 1-5 shows an example of address translation to a 4KB page using 32-bit PAE paging. The main differences are the introduction of another level in the paging structure hierarchy called the *page directory pointer table* and the fact that the paging structure entries are now 64 bits. Given these changes, the CR3 register now holds the physical address of the page directory pointer table.

Figure 1-6 shows the formats for the paging structure addresses that are used in 32-bit PAE paging. When PAE is enabled, the first paging table has only 4 (2^2) entries. The bits 31:30 from the virtual address select the page directory pointer table entry (PDPTE). The bits 29:21 are an index to select from the 512 (2^9) PDEs. If the PS flag is set, the PDE maps a 2MB page. Otherwise, the 9 bits extracted from bits 20:12 are selected from the 512 (2^9) PTEs. Assuming that all entries are valid, and the address is mapping a 4KB page, the final 12 bits of the virtual address specify the offset within the page for the corresponding PA.

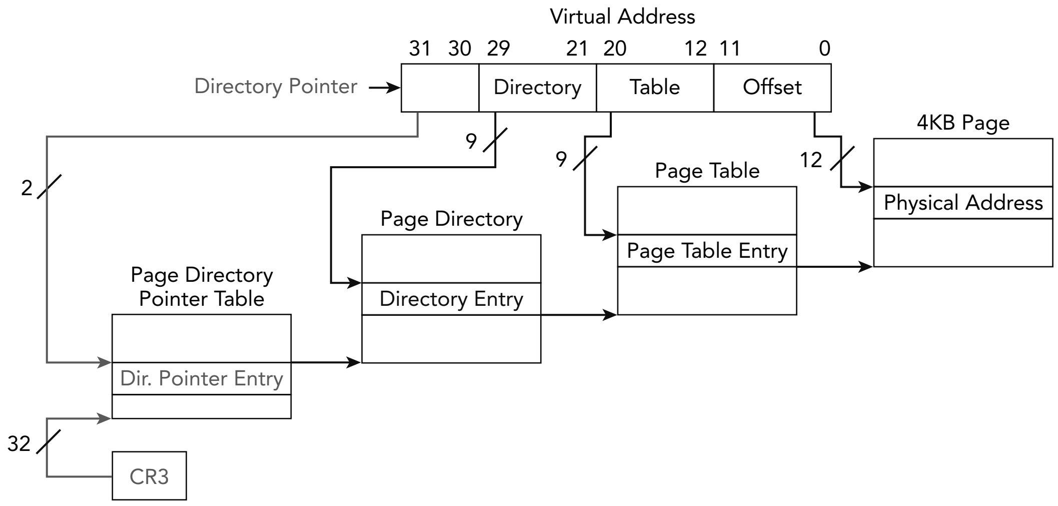

Figure 1-5: Address translation to a 4KB page using 32-bit PAE paging

5 5 4 4 4 4 4 4 4 4 4 4 3 3 3 3 3 3 3 3 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

PDPTE		CR[31:5]	V 3 1	V 3 0	0	0	0
PDE	PDPTE[51:12]	VA[29:21]	0	0	0		
2MB Page	PDE[51:21]	VA[20:0]					
PTE	PDE[51:12]	VA[20:12]	0	0	0		
PA	PTE[51:12]	VA[11:0]					

Figure 1-6: Formats for paging structure addresses used in 32-bit PAE paging

Intel 64 Architecture

The execution environment for the Intel 64 architecture is similar to IA-32, but there are a few differences. The registers highlighted in the IA-32 architecture still exist within Intel 64, but have been expanded to hold 64 bits. The most significant change is that Intel 64 can now support 64-bit linear addresses. As a result, the Intel 64 architecture supports a linear address space up to 2^{64} bytes. Note that the most current implementations of the architecture at the time of this writing do not support the entire 64 bits, only the 48-bit linear addresses. As a result, virtual addresses on these systems are in canonical format. This means that bits 63:48 are set to either all 1s or all 0s, depending on the status of bit

47. For example, the address `0xffffa800ccc0b30` has bits 63:48 set because bit 47 is set (this is also known as *sign-extension*).

It is also important for you to focus on the changes to memory management because they have a direct impact on memory forensics. The most important difference is that the Intel 64 architecture now supports an additional level of paging structures called *page map level 4* (PML4). All entries in the hierarchy of paging structures are 64 bits, and they can map virtual addresses to pages of size 4KB, 2MB, or 1GB. Figure 1-7 shows an example of address translation to a 4KB page using 64-bit/IA-32e paging.

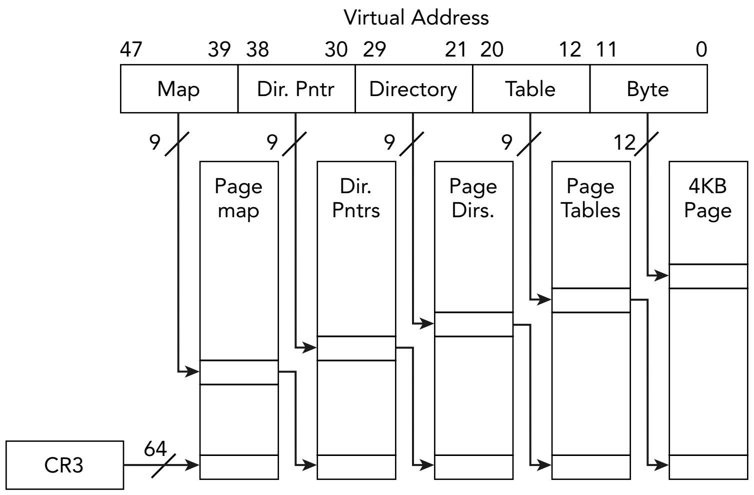

Figure 1-7: Address translation to a 4KB page using 64-bit/IA-32e paging

Figure 1-8 shows the formats for the paging structure addresses used in 64-bit/IA-32e paging. Each of the paging structures is composed of 512 entries (2^9) and is indexed by the values extracted from the following ranges taken from the 48-bit virtual address:

- Bits 47:39 (PML4E offset)
- Bits 38-30 (PDPTE offset)
- Bits 29:21 (PDE offset)
- Bits 20:12 (PTE offset)

If the PS flag is set in the PDPTE, the entry maps a 1GB page if it is supported. Similarly, if the PS flag is set in the PDE, the PDE maps a 2MB page. Assuming that all the intermediary entries are present, the final 12 bits specify the byte offset within the physical page.

If you are interested in the details of how the different paging structure entry flags affect memory forensics, you are encouraged to check out the Intel Manual and Volatility's `AMD64PagedMemory` address space.

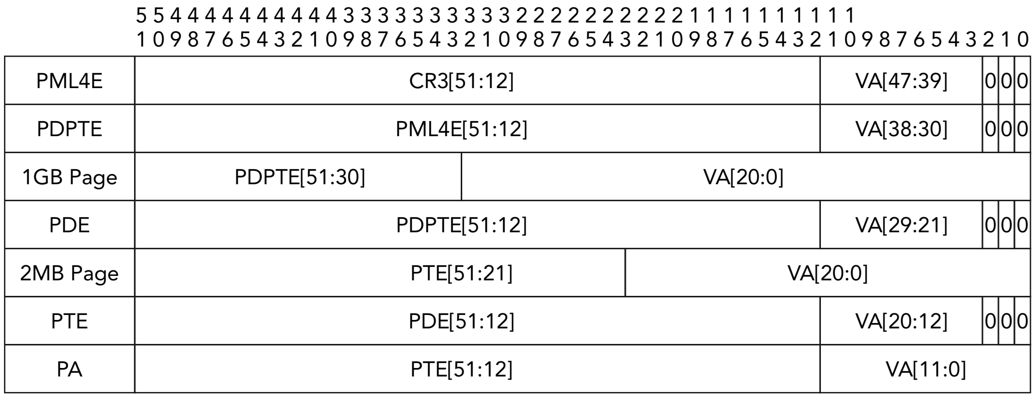

Figure 1-8: Formats for paging structure addresses used in 64-bit/IA-32e paging

Interrupt Descriptor Table

PC architectures often provide a mechanism for interrupting process execution and passing control to a privileged mode software routine. For the IA-32 and Intel 64 architectures, the routines are stored within the *Interrupt Descriptor Table* (*IDT*). Each processor has its own IDT composed of 256 8-byte or 16-byte entries, in which the first 32 entries are reserved for processor-defined exceptions and interrupts. Each entry contains the address of the *interrupt service routine* (*ISR*) that can handle the particular interrupt or exception. In the event of an interrupt or exception, the specified interrupt number serves as an index into the IDT (which indirectly references a segment in the GDT), and the CPU will call the respective handler.

After most interrupts, the operating system will resume execution where it was originally interrupted. For example, if a thread attempts to access a memory page that is invalid, it generates a page fault. The exception number 0xE handles page faults on x86 and Intel 64 architectures. Thus, the IDT entry for 0xE contains the function pointer for the operating system's page fault handler. Once the page fault handler executes, control can return to the thread that attempted to access the memory page. Operating systems also use the IDT to store handlers for numerous other events, including system calls, debugger breakpoints, and other faults.

WARNING

Given the critical role that the IDT performs for operating systems, it has been a frequent target of malicious software. Malicious software might try to redirect entries, modify handler code, add new entries, or even create entirely new interrupt tables. For example, Shadow Walker (`https://www.blackhat.com/presentations/bh-jp-05/bh-jp-05-sparks-butler.pdf`) hooked the page fault handler by modifying the IDT and was able to return "fake" pages to the caller.

An interesting paper regarding the use of IDT for rootkit and anti-forensic purposes is *Stealth Hooking: Another Way to Subvert the Windows Kernel* (`http://phrack.org/issues/65/4.html`). You can use the Volatility plugins `idt` (Windows) and `linux_idt` (Linux) for auditing the IDT.

Operating Systems

This section provides a general overview of the aspects of modern operating systems that impact memory forensics. In particular, it focuses on important features common to the three operating systems discussed in this book: Microsoft Windows, Linux, and Mac OS X. Although the topics might be familiar, this section discusses them within the context of memory forensics. Investigators familiar with operating system internals might choose to skip most of the material in this section or use it as a reference for topics covered in later chapters.

Privilege Separation

To prevent potentially malfunctioning or malicious user applications from accessing or manipulating the critical components of the operating system, most modern operating systems implement some form of user and kernel mode privilege isolation. This isolation attempts to prevent applications from affecting the stability of the operating system or other processes. The code associated with user applications (untrusted) executes in user mode, and the code associated with the operating system (trusted) executes in kernel mode.

This separation is enforced by the IA-32 processor architecture through the use of four privilege levels commonly referred to as *protection rings*. In most operating systems, kernel mode is implemented in *ring 0* (most privileged) and user mode in *ring 3* (least privileged). When the processor is executing in kernel mode, the code has unrestricted access to the underlying hardware, including privileged instructions, and to kernel and process memory regions (except on newer systems with SMEP, which prevents ring 0 execution of user pages). For a user application to access critical components of the operating system, the application switches from user mode to kernel mode using a well-defined set of system calls. Understanding the level of access that malicious code has gained can help provide valuable insight into the type of modifications it can make to the system.

System Calls

Operating systems are designed to provide services to user applications. A user application requests a service from the operating system's kernel using a system call. For example, when an application needs to interact with a file, communicate over the network, or spawn another process, system calls are required. As a result, system calls define the low-level API between user applications and the operating system kernel. Note that most applications are not implemented directly in terms of system calls. Instead, most operating systems define a set of stable APIs that map to one or more system calls (for example, the APIs provided by `ntdll.dll` and `kernel32.dll` on Windows).

Before a user application makes a system call, it must configure the execution environment to pass arguments to the kernel through a predetermined convention (for example, on the stack or in specific registers). To invoke the system call, the application executes a software interrupt or architecture-specific instruction, which saves the user mode register context, changes the execution mode to kernel, initializes the kernel stack, and invokes the system call handler. After the request is serviced, execution is returned to user mode and the unprivileged register context is restored. Control then returns to the instruction following the system call.

Because it is such a critical bridge between user applications and the operating system, the code used to service system call interrupts is commonly intercepted by security products and targeted by malicious software. Later in the book, you will learn how to use memory forensics to detect modifications made to this critical interface on Windows, Linux, and Mac systems.

Process Management

A *process* is an instance of a program executing in memory. The operating system is responsible for managing process creation, suspension, and termination. Most modern operating systems have a feature called *multiprogramming*, which allows many processes to appear to execute simultaneously. When a program executes, a new process is created and associated with its own set of attributes, including a unique process ID and address space. The *process address space* becomes a container for the application's code, shared libraries, dynamic data, and runtime stack. A process also possesses at least a single thread of execution. A process provides the execution environment, resources, and context for threads to run. An important aspect of memory analysis involves enumerating the processes that were executing on a system and analyzing the data stored within their address spaces, including passwords, URLs, encryption keys, e-mail, and chat logs.

Threads

A *thread* is the basic unit of CPU utilization and execution. A thread is often characterized by a thread ID, CPU register set, and execution stack(s), which help define a thread's execution context. Despite their unique execution contexts, a process's threads share the same code, data, address space, and operating system resources. A process with multiple threads can appear to be simultaneously performing multiple tasks. For example, one thread can communicate over the network while another thread displays data on the screen. In terms of memory forensics, thread data structures are useful because they often contain timestamps and starting addresses. This information can help you determine what code in a process has executed and when it began.

CPU Scheduling

The operating system's capability to distribute CPU execution time among multiple threads is referred to as *CPU scheduling*. One goal of scheduling is to optimize CPU utilization as threads switch back and forth between waiting for I/O operations and performing CPU-intensive computation. The operating system's scheduler implements policies that govern which threads execute and how long they execute. Switching execution of one thread to another is called a *context switch*.

An execution context includes the values of the CPU registers, including the current instruction pointer. During a context switch, the operating system suspends the execution of a thread and stores its execution context in main memory. The operating system then retrieves the execution context of another thread from memory, updates the state of the CPU registers, and resumes execution where it was previously suspended. The saved execution context associated with suspended threads can provide valuable insight during memory analysis. For example, it can provide details about which sections of code were being executed or which parameters were passed to system calls.

System Resources

Another important service that an operating system provides is helping to manage a process' resources. As previously mentioned, a process acts as a container for system resources that are accessible to its threads. Most modern operating systems maintain data structures for managing the resources that are actively being accessed, which processes can access them, and how they are accessed. Examples of operating system resources that are typically tracked include processes, threads, files, network sockets, synchronization objects, and regions of shared memory.

The type of resources being managed and the data structures being used to track them often differ between operating systems. For example, Windows leverages an object manager to supervise the use of system resources and subsequently stores that information in a handle table. A handle provides the process with a unique identifier for accessing and manipulating system resources. It is also used to enforce access control to those resources and track their usage. Linux and Mac both use file descriptors in a similar manner. Later in the book, we describe how to extract this information from the handle or file descriptor tables and how to use it to gain insights into that process' activity.

Memory Management

Memory management refers to the operating system's algorithms for managing the allocation, deallocation, and organization of physical memory. These algorithms often depend on the previously discussed hardware support.

Virtual Memory

Operating systems provide each process with its own private virtual address space. This abstraction creates a separation between the logical memory that a process sees and the actual physical memory installed on the machine. As a result, you can write programs as if they have access to the entire address space and in which all ranges are memory resident. In reality, some pages of the address space might not be resident. Behind the scenes, the memory manager is responsible for transferring regions of memory to secondary storage to free up space in physical memory. During execution, the memory manager and the MMU work together to translate the virtual address into physical addresses. If a thread accesses a virtual address that has been moved to secondary storage, that data is then brought back into physical memory (typically via page fault). This interaction is represented in Figure 1-9.

The actual size of the virtual address space often depends on the characteristics of the hardware and operating system. Operating systems frequently partition the range of accessible addresses into those addresses associated with the operating system and those that are private to the process. The range of addresses reserved for the operating system is generally consistent across all processes, whereas the private ranges depend on the process that is executing. With the support of the hardware, the memory manager can partition the data to prevent a malicious or misbehaving process from reading or writing memory that belongs to kernel memory or other processes.

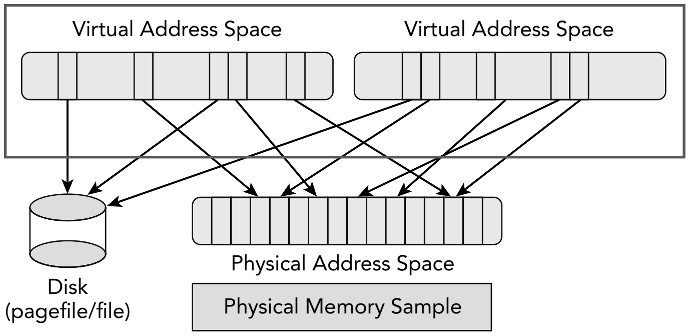

Figure 1-9: Illustration of multiple virtual address spaces sharing memory and secondary storage

Demand Paging

The mechanism that is commonly used to implement virtual memory is *demand paging*, which is a memory management policy for determining which regions are resident in main memory and which are moved to a slower secondary storage when the need arises. The most common secondary storage is a file or partition on an internal disk, referred to as the *page file* or *swap*, respectively. A demand paging implementation attempts to load only the pages that are actually needed into memory as opposed to entire processes.

Demand paging relies on a characteristic of memory usage known as *locality of reference*, which is based on the observation that memory locations are likely to be frequently accessed in a short period time, as are their neighbors. Ideally, demand paging reduces the time it takes to load a process into memory and increases the number of processes that are memory resident at any one time. To improve performance and stability, an operating system's memory manager often has a mechanism for designating which regions of memory are paged versus those that must remain resident.

The memory manager typically tracks which pages are memory resident and which are not in the previously discussed paging data structures. If a thread attempts to access a page that is not resident, the hardware generates a page fault. While the hardware generates the page fault, the operating system leverages state information encoded in the paging structures to determine how to handle the fault. For example, the page might be associated with a region of a file that had not been loaded into memory, or the page might have been moved to the page file.

Demand paging provides substantial benefits to the operating system and is transparent to running applications. As you will see in later chapters, it does add some complexity to memory forensics because some pages might not be memory resident at the time the memory sample is collected. Under certain circumstances, it is possible to combine non-memory-resident data found on disk with the data stored in memory to provide a more complete view of virtual memory.

Shared Memory

The previous sections discussed how process address spaces are isolated from each other to improve system security and stability. However, modern operating systems also provide mechanisms that allow processes to share memory. You can view shared memory as memory that is accessible from more than one virtual address space. For example, Figure 1-10 shows that Process A and Process B have regions of their private virtual address space that map to common pages in physical memory. One common use for shared memory is to provide an efficient means of communication between processes. After a shared region is mapped into virtual address spaces, processes can use the region to exchange messages and data.

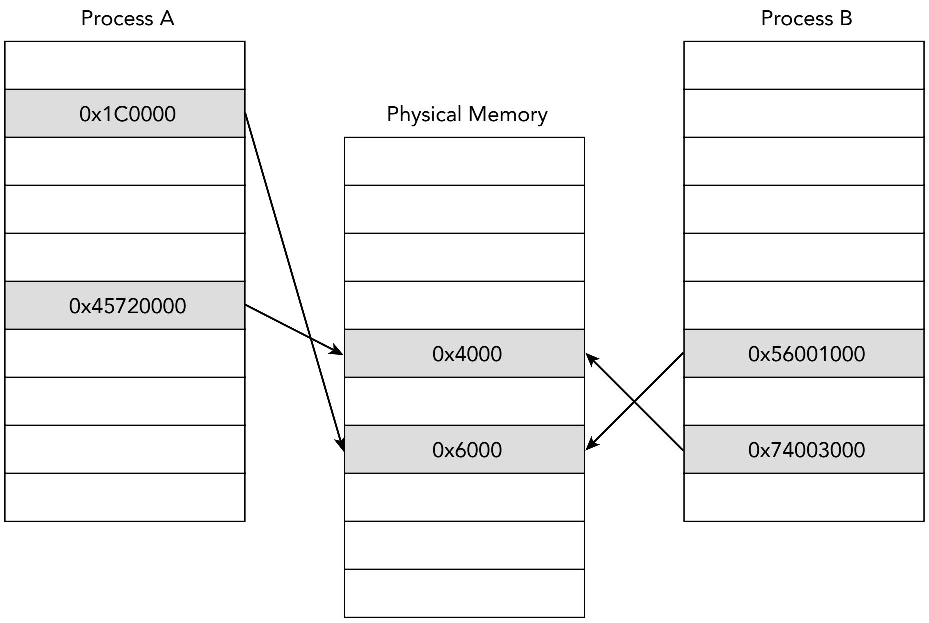

Figure 1-10: Example of shared memory mappings between two processes

Shared memory is also commonly used to conserve physical memory. Instead of allocating multiple physical pages that contain the same data, you can create a single instance of the data in physical memory and map various regions of virtual memory to it. Examples include shared or dynamic libraries that contain common code and data. In these cases, the shared pages are typically mapped as *copy-on-write*, which allows the memory manager to defer making a private copy of the data within a process' address space until the memory has been modified. After the page is written to, the memory manager allocates a private copy of that page with the associated modifications and updates the virtual memory mappings for that process. The other processes are unaffected and still map to the original shared page.

Both shared memory and copy-on-write mappings are frequently encountered during memory forensics because malicious software often attempts to modify the code of shared libraries to hijack the flow of execution. In Chapter 17, you see an example of how to spot discrepancies by comparing the data shared between multiple processes.

Stacks and Heaps

The user address space is typically divided into a number of regions. The *stack* region holds the temporary data associated with executing functions. The data in this region is stored in a data structure called a *stack frame*. Each frame includes information, such as the function parameters, local variables, and the information required to recover the previous stack frame. When a thread is executing, stack frames are stored (pushed) when calling a function and removed (popped) when returning from a function. Because a process can execute in either kernel mode or user mode, operating systems typically use a separate stack for the functions executed within each mode.

Analysis of remnant and active stack frames are extremely useful during memory forensics because they provide valuable insight into which code was being executed and what data was being processed. For example, keys can be passed to encryption routines, stolen data from the computer (keystrokes, file contents) can be sent to functions for exfiltration, and a number of other possibilities. During malware analysis, stack frames can be used to infer what part of the malware was active and what parts of the system the malware was interacting with.

NOTE

Carl Pulley wrote a stack unwinding plugin for Volatility named `exportstack` (`https://github.com/carlpulley/volatility`). It integrates with Microsoft's debugging symbols so that it can properly label addresses and associate them with API function names. Edwin Smulders wrote a similar plugin named `linux_process_stack` (`https://github.com/Dutchy-/volatility-plugins`) for analyzing stacks in Linux memory dumps.

The application's data that needs to be dynamically allocated is stored within the region called the *heap*. Unlike data allocated on the stack, which persists only for the scope of a function, the data allocated within the heap can persist for the lifetime of the process. A heap stores information whose length and contents may not be known at compile time.

Applications can allocate memory regions on the heap as they are needed and then deallocate them after use.

The operating system might also have regions of memory that are dynamically allocated within kernel mode. For example, Windows creates paged and nonpaged regions within the kernel that are referred to as *pools*. Common examples of interesting data that you can find in the heap include data read from files on disk, data transferred over the network, and input typed into a keyboard. Due to the nature of data stored within it, the heap can provide valuable evidence during forensics investigations. Because the data can be application dependent, manual analysis might be required, such as viewing data with a hex editor or by extracting strings for further examination.

File System

We previously discussed how the memory management subsystem leverages secondary storage to free up main memory. Operating systems also use secondary storage to manage persistent data objects that a user wants to access for a timeframe longer than the lifetime of any particular process. Unlike volatile main memory, secondary storage is typically composed of nonvolatile block devices such as hard disks. The collection of data structures that allow an application to perform primitive operations on the stored data is called a *file system*. File system forensics involves finding files or content of interest, recovering file artifacts (deleted, fragments, hidden), and leveraging temporal metadata such as timestamps to reconstruct the events of an incident.

Although file systems have historically been one of the most common sources of digital evidence, general file system forensic analysis is not a focus of this book. This book discusses file system artifacts that you find in volatile storage, main memory artifacts that you find within the file system, and how you can combine these types of data to provide a more comprehensive view of the state of a system. For example, data stored in files and the directory structures must be loaded into memory when they are needed. The operating system also caches frequently accessed data in main memory to reduce the overhead associated with repetitively querying slower secondary storage.

Previous sections discussed how the memory management subsystem uses demand paging and shared memory to optimize memory usage. Most modern operating systems also support memory mapped files, which enable files or portions of files to map into the virtual address space. After files map into memory, you can access and modify them in the same manner as traditional in-memory data structures such as arrays. As a result, the optimized functions of the operating system are responsible for transparently handling the disk I/O, in which the file becomes the backing store. Pages of file data are read into memory when addresses within the page are accessed, and regions of file data can be easily shared among processes.

Investigators can leverage information about cached file data to help triage and provide context about recently accessed and frequently accessed data. The characteristics can also provide insight into which users or processes were accessing the data. By comparing cached data with the data stored on disk, investigators can also identify modifications made to memory-resident data. Additionally, during file system analysis, investigators might find memory artifacts in crash dumps or hibernation files that can provide insight into previous states of the system. Thus, although this book does not cover general file system forensics, a familiarity with file systems is useful.

I/O Subsystem

One of the major services that an operating system offers is managing and providing the interface to peripheral input and output (I/O) devices. The I/O subsystem abstracts the details of these devices and enables a process to communicate with them using a standard set of routines. Many operating systems generalize the interface to devices by treating them as files. Because you cannot predict the type of devices that people will connect to the system, operating systems use kernel modules called *device drivers* as a mechanism for extending the capabilities of the kernel to support new devices.

Device Drivers

Device drivers abstract away the details of how a device controls and transfers data. Device drivers typically communicate with the registers of the device controller. Although some CPU architectures provide a separate address space for I/O devices and subsequently require privileged I/O instructions, other architectures map the memory and registers of I/O devices into the virtual address space. This is typically referred to as memory mapped I/O. As you see in later chapters, software commonly abuses device drivers to modify the state of the system.

Operating systems also use device drivers to implement virtual, software-only devices. For example, some operating systems provide a representation of physical memory via a software device (for example, `\Device\PhysicalMemory` on Windows). This device interface has been commonly used to collect forensic samples of physical memory. Note that device memory and registers might also be mapped into memory, which can have interesting consequences for memory acquisition (see Chapter 4).

I/O Controls (IOCTLs)

I/O Control (*IOCTL*) commands are another common mechanism for communicating between user mode and kernel mode. Although system calls provide a convenient interface for accessing the fixed services provided by the operating system, user applications might need to communicate with a variety of peripheral devices or other operating system components. IOCTLs allow a user application to communicate with a kernel mode device driver. They also provide a mechanism for third-party hardware devices and drivers to define their own interfaces and functionality.

As with system calls, kernel-level malware might hook IOCTL functions in order to filter results or modify control flow. Malware has also used IOCTL handlers to communicate between user mode and kernel mode components (for example, to request that a kernel component elevate privileges, disable a service, or modify firewall settings). Memory forensics can detect modified or unknown IOCTLs and provide valuable insight into how attackers leverage them.

Summary

Now that you're familiar with the primitive hardware and software concepts of the digital environment, you can move on to exploring the types of data you'll encounter throughout forensic analysis of digital media, such as RAM. Remember that the information shared in this chapter is not specific to any one operating system—it applies to Windows, Linux, and Mac OS X. Thus, as you're reading the rest of the book, you may need to refer back to this chapter to refresh your memory on keywords that you see us using in other discussions.